Conformity Issues of Ada Tasking and the Effects of Ada Generics on Object Code Size

THESIS

Robert H. Tippett
Captain, USAF

AFIT/GCS/ENG/89M-3

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/89M-3

Conformity Issues of Ada Tasking and the Effects of Ada Generics on Object Code Size

THESIS

Robert H. Tippett
Captain, USAF

AFIT/GCS/ENG/89M-3

Approved for public release; distribution unlimited

AFIT/GCS/ENG/89M-3

CONFORMITY ISSUES OF ADA TASKING AND

THE EFFECTS OF ADA GENERICS

ON OBJECT CODE SIZE

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

Robert H. Tippett, B.S., B.A.

Captain, USAF

March 1989

Table of Contents

Preface

The purpose of this thesis effort was basically threefold. First, tests were developed and applied to a variety of Ada compilers for testing the fairness of scheduling among equal priority concurrent processes running on a single processor machine. Second, Ada test code was developed to confirm the ability of programmers to construct an implied priority scheme using nested conditional entry calls. Third, a series of generic programs and their explicitly coded counterparts were constructed to evaluate the differing effects on object code size between each method of coding.

While these three topics seem somewhat independent, the overall effort of this research was undertaken as a compiler testing research topic. In that light, the continuity of the work seems somewhat more plausible. Hence, the reader should look at the whole as Ada compiler testing research and each part as a specific problem in the area of Ada compiler testing.

I want to thank my thesis advisor, Major Jim Howatt, for his patience and understanding during the course this research. Also, I want to thank my committee members, Major Donna Herge and Capt Will Bralick for their suggestions and assistance.

Most of all, I want to thank my wife Mary for her steady support while I was frequently distracted by my work at AFIT. I was able to complete this program because of her steadfast support throughout the long months of study.

List of Figures

Figure

List of Tables

Abstract

The main purpose for this research is Ada compiler testing. Specifically, the thesis was intended to explore three areas of concern. First, tests were designed and implemented to try to determine the fairness of Ada task scheduling algorithms within the run-time system of Ada compilers. Next, an algorithm for using the Ada conditional entry call in implied priority schemes was developed. The algorithm was then coded and verified for correctness. The third and final part of this research explored the effect of using Ada generics on object code generation. Six Ada programs were developed in both generic and explicit forms to determine the extent of object code inflation when using generics.

Results of the work show that many individual compilation systems approach the problem of task scheduling differently. The evidence gathered showed that, for the tests used in this part of the work, the compilers were, in general, not fair to equal priority tasks. While the tests were structured to ensure that all tasks had equal priorities and equal duties to perform, they were also designed with disregard for timing requirements on the rendezvous. Hence, the unfairness of a compilation system may be traceable to the large overhead involved in using the rendezvous.

It was determined that the Ada conditional entry call can be nested to provide an implied priority scheme. Only three levels

of priority were implemented for this work, but that could easily be extended to the range allowed by the compilation system.

The final area of research for this thesis explored Ada generics. It was found that using generics does not necessarily mean the object code generated will be larger than that of the explicitly typed version of the same program. In fact, in some instances, the explicit version generated more object code than the generic. Therefore, the commonly held notion that generics produce more object code is false. It is correct to say that object code size depends on the source code, the compiler and the particular machine.

Conformity Issues of Ada Tasking and the Effects of Ada Generics on Object Code Size

I. Background

Since the late 1960s and early 1970s, software development costs have spiraled upward at a critical rate (Shumate 1984:10-11; Booch, 1987:6-12; Woffinden, 1987). The impact of these spiraling costs continues to plague current development efforts in the private sector as well as in the Department of Defense (DoD)(Port et al, 1988:142-154).

Once the problem of increasing software development costs became well documented by DoD, a solution in the form of a standardized programming language (Fisher, 1978:24-33), was put forth by components of DoD software development organizations. The new programming language was to incorporate all the best features of modern languages. Also, it was to be designed with particular emphasis on software engineering and code maintenance. In other words, the new language was to provide a syntax and semantic structure that necessarily enforced good engineering practices in the development of new software. Additionally, it was to be self-documenting (human readable) so life-cycle maintenance costs could can be trimmed (Elbert, 1986:4-6; Booch, 1987:28-42).

In May 1979, a contract was awarded to the French company, Honeywell/Honeywell Bull, for the development of the new programming language. The new language, Ada, came into being in

July 1980 and included many features of the better languages of the day (Booch, 1987:13-24; Gehani, 1987:xiii-xvii).

As with any new approach to problem resolution, a certain amount of resistance to Ada was expected in one form or another. The use of Ada in programming shops throughout the industry met with its share of resistance. Initially, most of the resistance was in the form of sharp criticisms about the language's size and complexity. World renowned software development experts lined up to say the language was far too large and extremely complex (Dykstra, 1978:21-26; Hoare, 1981:75-83).

Some experts even went as far as writing a language and reference manual that completely satisfied the IRONMAN specification (DoD, 1977), yet remained comparatively small, simple and concise (Shaw et al. 1978;36-58). This effort was in response to the proposed languages submitted for the "fly-off" of the contract. The authors of the TARTAN Ada challenged the four companies, who were competing for the contract, to match the size and lack of complexity they had achieved in their implementation of Ada. Still others felt that Ada could be effectively down-sized to make it more appealing to the user community (Ledgard, 1982:121-125).

More recently, other prominent experts have stepped forward to defend Ada's size and complexity by virtue of the problem space for which it was designed (Wichmann, 1984:98-103; Elbert, 1986:1-5; Booch 1987:Ch 1; Ada Board, 1988). The problem space, namely large, real-time, embedded systems, is by nature large and

complex and these authors strongly defend Ada's size and complexity.

Regardless of the arguments put forth on either side of the issue, the relative worth of Ada as a viable answer to DoD's software woes is academic. DoD now mandates the use of Ada in Mission Critical Computer Resources (DoD, 1987:42-44). Hence, it is now not a matter of whether to use Ada, but more a matter of how to use it to solve problems in the large, real-time, embedded systems of many DoD organizations.

Relevant Issues

Two of the more modern features of powerful programming languages are concurrent processing and abstraction. Each of these features exists to provide the framework from which the software engineer constructs programs that adhere to accepted principles of software development. Additionally, these features allow users to exploit new system capabilities and new development methodologies such as distributed processing and object oriented design.

The capability for concurrent processing enhances a language's usefulness because many modern programming applications require simultaneous processing of data on a real-time basis (Gehani, 1984:1-5; Elbert, 1986:376-380).

Many prominent software engineers place abstraction at the top of the list of software engineering principles (Jensen, 1979:275; Elbert, 1986:5-9; Booch, 1987:31). Therefore, if

abstraction is well supported by the programming language, then software solutions to problems become more manageable because the designer is not forced to spend inordinate amounts of time on the details of machine implementation of the particular language.

Ada, with all its largeness and complexity, supports both concurrent processing and abstraction. Concurrent processing, the ability to simultaneously execute multiple paths of code from within a single program, is available in Ada via its tasking facility (DoD, 1983:Ch 9). Ada tasking can be used on multiple processor systems to achieve true concurrent processing or on a single processor system to implement pseudo-concurrency (pseudo-concurrency means simultaneous execution of statements in a program is impossible since only one processor exists--the appearance of concurrency is given by virtue of the speed of the CPU).

Abstraction is available in Ada through strong data typing and the use of generics. Strong data typing (predefined and user-defined) and generics alleviate constraints caused by rigid sets of pre-defined data types and data structures. Abstraction provides capabilities which give the programmer countless options in regard to data types and data structures. The implementation details of a particular type or structure are raised to a sufficiently high level so that the programmer does not generally need to concern himself with the inner workings of the machine architecture. Thus, more time can be spent on solving the

problem at hand instead of wrestling with the details of the particular hardware.

Concurrent Processing. Concurrent processing is a necessary feature for most real-time embedded systems. Real-time embedded processing means the data processing to be accomplished is part of a larger system effort to process all data within certain minimal time constraints. In a multiple processor system (Figure 1), central processing unit (CPU) time for a process is generally available by virtue of dedicated processors for each sub-process within the applications program.

P0 through Pn, shown in Figure 1, are separate processors which perform specific functions in the overall system. Specific sub-processes are allocated or scheduled to specific processors within the overall system in order to accomplish the real-time system requirements. The processor controller, P0, has the explicit function of overseeing operation of the subordinate processors in the system.

In a single processor system (Figure 2), however, the CPU time must be allocated to each sub-process (SP) in turn. Hence, in the single processor system used for real-time processing, sub-processes (shown in Figure 2 as SP1..SPn) with higher priority for completion may interrupt lower priority sub-processes and cause the CPU to become exclusively dedicated to completing the higher priority jobs. Obviously, this creates a need for some sort of scheduling algorithms to ensure all sub-

processes are allocated some CPU time based on a predetermined scheduling algorithm.

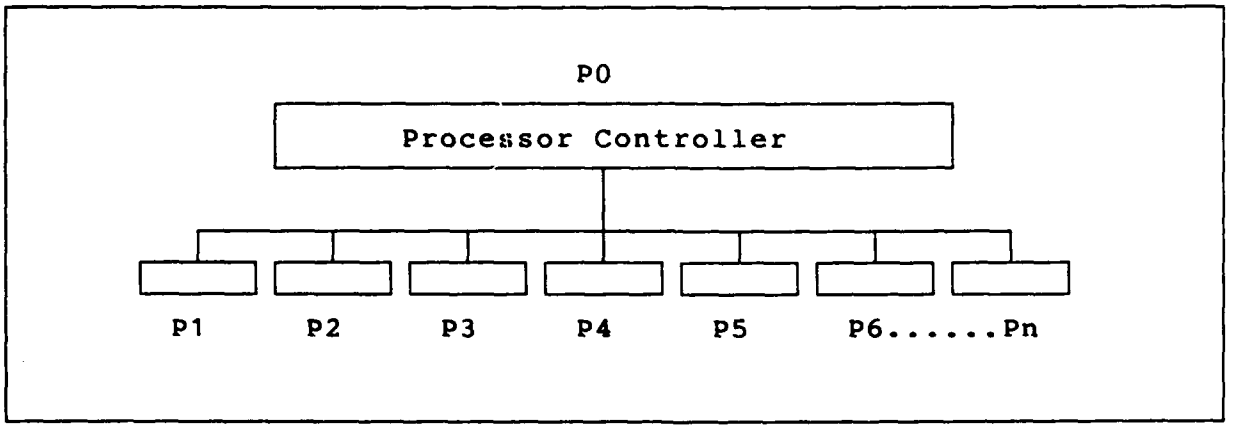

Figure 1. Multiple Processor Computer System

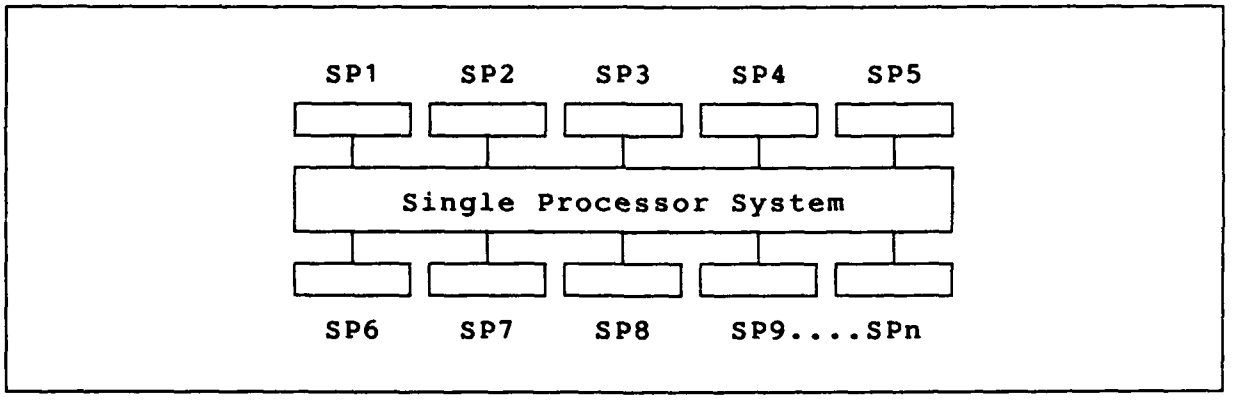

Figure 2. Single Processor Computer System

The scheduling algorithms used by the Ada run-time environment should provide some percentage of the total processor time for each priority sub-process. If the scheduling algorithms do not allow lower priority sub-processes to execute, starvation will occur and the overall system requirements will not be met.

Abstraction. Abstraction, as mentioned above, is present in Ada as strong typing of all data types. Each element identified for use must be declared prior to its use within the context of the language. Furthermore, data types and structures may be specified in a generic sense. For generics, the particular details of a structure are defined, but the data elements themselves can vary. Specification of a generic structure in Ada allows the software engineer to develop one piece of code which will perform the same task on many different data types.

For instance, programmers spend significant amounts of time writing routines which sort particular groups of elements in a certain order. Generic programs are open ended in the sense that, given the characteristics of the data elements to be sorted, they can sort uniquely different sets of data elements (each element in the set must be the same type) without modifying the body of the sort routine. This is a great time saver since the user can merely instantiate the generic and process the different sets of data elements without writing a complete new sort routine.

Concurrent processing and abstraction are two features of Ada which allow the software engineer to apply accepted software engineering principles to software development. Each of these features of Ada still requires many refinements as the language becomes an integral part of the mainstream development efforts within DoD and private industry. As such, many topics for research and development have been put forth to make Ada better in the coming years.

Problem

There are wide ranges of areas for research within the Ada language. Three of those areas, task scheduling, conditional entry calls, and generics are briefly discussed in the following sections.

Fairness of Task Scheduling. Current understanding of the limitations of Ada tasking centers on the interpretation of the Ada Language Reference Manual (LRM). The method of scheduling tasks with equal priorities is not determined in the language (DoD, 1983:9-17). The actual implementation of particular scheduling algorithms is normally proprietary to the developer of a specific Ada compiler. As such, it is hard to discern the actual structure and implementation details of a given compilation system.

Since Ada is designed to be machine independent, a method to test different compilers for task scheduling fairness can be constructed (Evans, 1987). The test can then be used to determine whether a particular Ada compiler provides an equitable amount of CPU time to equal priority tasks.

Implied Priority Using Nested Conditional Entry Calls. Another issue of concern in Ada tasking (Evans, 1987) is the ability of a programmer to impose a priority scheme on tasks through the use of the conditional entry call (LRM, 1983: Ch 9.7.2). By using nested conditional entry calls, programmers should be able to specify the sequence in which tasks run.

It might be possible to nest the entry calls in a controller task of some kind. Then, given the proper structure of the code following the else option in the conditional entry call, an artificial priority scheme could exist over the scope of the nesting. To realize the priority scheme, the alternative after the entry call should be just another conditional entry call. Unique priorities should be available through the entry points of the controller task.

Generics versus Explicitly Typed Programs. Finally, in the area of abstraction, there is little or no information published about the effects of using generic instantiations in place of explicitly typed routines within an Ada program. At the center of the issue is the question: given the space limitations involved in some embedded applications programs (avionics software in particular), does the size of object code generated by generic programs force the designer to use explicitly typed routines?

Stated more simply, the purpose of this portion of the research is to determine the effects on object code size of using generic programs, instead of the explicitly coded versions.

Problem Summary

Three independent problems have been put forth for resolution by ASD/SCEL (Evans, 1987). They are as follows:

1. Is task scheduling on a single processor system fair, based on equal priority tasks?
2. Can nested conditional entry calls in select statements be used to produce an implied priority scheme?
3. How do generic programs compare with explicitly typed programs with respect to object code size?

Current Knowledge

Current knowledge about concurrent processing in Ada is fairly extensive (Gehani, 1984; Elbert, 1986; Booch, 1987; Shumate, 1988). However, specific knowledge concerning task scheduling algorithms is somewhat sparse, since these algorithms are proprietary to the vendors of compilers. Each vendor may implement the scheduling algorithms differently to achieve the required results. The vendor is given free rein to develop all aspects of the Ada compilation systems. Once it is developed, however, it must pass validation tests to be approved for use by DoD components (Goodenough, 1981).

The Ada Compiler Validation Capability (ACVC) Implementors' Guide is a document which enumerates tests for new Ada compilers. Currently no tests exist in the document to determine the fairness of task scheduling in a single processor system (Goodenough, 1986:Ch 9). Tests will be developed during this research which may be used to determine the fairness of Ada

compiler task scheduling algorithms.

Some aspects of the conditional entry call have not been addressed in the ACVC. For instance, is it possible to use the conditional entry call to impose an implied priority scheme over several levels of nesting? If such is the case, then a program which provides this feature can be developed and used. The program would process all entry calls based on the priority of the particular entry point called. Again, no tests exist in the ACVC Implementors' Guide to test this aspect of tasking (Goodenough, 1986:Ch 9).

Although there is much speculation among software engineers concerning the generation of large amounts of object code while using generics in Ada, little is published on the subject. Since the size of object code correlates directly to memory requirements of the underlying machine, many professionals agree that use of generics in systems that are memory space sensitive (aircraft avionics to name one) is generally not good practice. This mind set is based on speculation and some limited evidence (Lyon, 1988; King, 1988). Little hard knowledge exists on the effects of using generics versus explicitly typed programs and this portion of research will establish a knowledge base across several Ada compilers.

Assumptions

Assumptions made for this research effort are based on the availability of compilers to run these various test suites. Numerous compilers are available through the Air Force Institute of Technology's (AFIT) computer center. Several more are available from the ADA Language Control Facility (ASD/SCEL). The exact number used during the research will depend on implementation problems encountered as the effort progresses. Ideally, the code generated for this research effort will be extremely portable and few machine/language interfacing problems are expected.

Scope

As mentioned above, the scope of the proposed study is limited to developing and using methods for determining the fairness of task scheduling, verifying the performance of nested conditional entry calls against the desired result, and understanding the effects of generic code segments versus explicitly typed segments with respect to generation of object code. Other issues concerning concurrent processing, select statements, and abstraction in Ada are not addressed.

Standards

Since the LRM (DoD, 1983) is the standard that defines Ada, it will be the source document for validity of Ada code written as part of this research. Additionally, the ACVC Implementors'

Guide (Goodenough, 1986) will be used for determining the types of tests which already exist and areas where tests may be needed.

Approach/Method

The specific approaches used to solve each of the above problems are given in detail in Chapters 3, 4 and 5 respectively.

Materials and Equipment

All materials and equipment necessary for completing this work were available through AFIT and ASD/SCEL.

II. Literature Review

Introduction

The overall purpose of this chapter is to discuss the literature reviewed for the research of this thesis. The information presented in this chapter is taken from many sources and was gathered in order to gain a more complete understanding of the problems presented in Chapter 1. Considerable amounts of literature exist concerning nearly every aspect of Ada. Topics of particular importance to this thesis effort are knowledge concerning the Ada Run-Time Environment (RTE), Ada concurrency and task scheduling, the use of Ada conditional entry calls, and object code generation by Ada generics. Additionally, this research effort falls under the heading of compiler testing so a discussion of current compiler testing literature is included.

The material included in this chapter provides a framework for understanding of the topics at hand. While the information on RTEs and software testing may seem somewhat misplaced, it is included for two reasons. The first reason for discussing these areas is completeness. Research into the area of Ada compilers and programming could not proceed very well without a basic understanding of the compilation systems and software testing.

Second, all these topics are pertinent to this research in either a direct or an indirect sense. While software testing is not directly related to any of the three problems of this thesis, a general understanding of it is necessary. Hence, the five areas

mentioned in the first paragraph were studied to gain a basic framework of knowledge about RTEs, Ada programming and software testing.

Run-Time Environments

This discussion of a general framework of Run-Time environments (RTE) is given to define the notion of a RTE as it pertains to this thesis. Also, the distinction between the Ada RTE and the Ada Run-Time System (RTS) is discussed to show the subtle differences in the two. The main source for this discussion is an article from the Ada Run-Time Environment Working Group of SIGAda (AREWG, 1988:51-68).

Run-Time Environments in General. RTEs are a consequence of the evolution of the man/machine interface of computer systems. When computers were still fairly new, users interacted with them in a very fundamental manner. The user simply wrote each and every software application from scratch, based on the underlying architecture of the machine at hand. This was a sufficient mode of operation when the programs were small and the user was the only person involved in developing the application. When larger programs and development teams began to emerge, several new ideas about the process of writing software became evident.

First, developers began to adopt certain conventions for writing applications software. This, then, provided a baseline level of reliability and interoperability between applications when the conventions were used. The conventions were generally

concerned with utilization of the underlying machine architecture and with standard sub-routine interfaces within applications programs.

There was also a recognition that the bare machine did not, in and of itself, provide enough abstraction in the representation of some of the more common data structures. Hence, conventions for the development of the most common data structures were adopted to allow the details of implementation to be suppressed. With these conventions in force, the logical evolution of these ideas lead to pre-written subroutines that could be included in applications programs.

These ideas, namely coding conventions and pre-written subroutines, formed the basis for the evolution of modern RTEs. With time, several results of using the RTEs emerged. Development of applications programs proceeded with much less concern for the machine architectural structure. Consequently, resources formerly spent on interfacing with the architecture, went toward writing the best possible applications program.

Eventually, the process of supporting the above concepts became automated. These automated tools became the present day operating systems for general purpose computers (executives for embedded computers) and programming language compilers.

Operating systems (and executives) provide the necessary subroutines to control utilization of the machine resources. The program language compilers provide the coding conventions for

data structures and the critical interface to the services of the operating system.

Overall, the use of these automated features offers the developer a much higher level of abstraction for the process of writing applications programs. They also offer a consistent interface to different users of the same environment. The abstraction of the programming process and the consistent environmental interface aid the applications developer, since significantly less time is spent on the details of the machine architecture and inconsistencies of the environmental interface.

The present day configuration of the general RTE is a hybrid relationship between the underlying machine capabilities, the operating system (or executive) and the program language compiler. The RTE is dynamic in the sense that it can be different for different combinations of the three components just mentioned. For any given combination of the components, a change in one may imply corresponding changes to the other two.

If, for example, the compiler used in a system is replaced with a different manufacturer's product, a different set of machine capabilities and operating system services may be required. The existing machine and operating system may or may not be able to meet the new requirements. Likewise, changes to one of the other components can cause the structure of the system to be adjusted to support the new configuration. In general, using the RTE involves a cost benefit analysis. Namely, is the degradation in the performance of a particular application

(caused by carrying the necessary elements of the RTE along) offset by the convenience of using the RTE? In today's world of program development, the answer is generally yes.

There are still specialized applications, particularly in the embedded, real-time arena, which do not lend themselves to the use of the RTE. The reasons for this are, generally, associated with timing constraints that are not maintainable within the RTE. In general, however, the RTE approach to applications programming is widely accepted and practiced.

To this point in the development of the RTEs, the specific tasks assigned to the three component parts of the RTE were clearly delineated. Responsibility of each component within the RTE could change, as mentioned above, but once the RTE was adjusted to meet the new configuration requirements, each part performed a specified number of services.

Ada Run-Time Environments. With the arrival of Ada on the scene, the boundaries between the responsibilities of the RTE component parts became somewhat blurred. Since Ada includes concurrent programming and memory management capabilities, both of which require the services of the operating system and compiler, there needed to be more flexibility within the language itself. Additionally, Ada does not require a specific operating system or executive in order to generate applications programs. Hence, the compilation system within the Ada RTE assumes the burden of providing all support for developing applications programs.

The Ada RTE is composed of the same three components of other general RTEs. The difference is that the compilation system in the Ada RTE chooses elements from the available subroutines in the Ada Run-time library (RTL). The set of subroutines from the RTL used by any particular Ada application program is referred to as the run-time system for that application. The RTS is the subset of the available subroutines needed to successfully execute a particular application.

The compilation system is governed by the syntax and semantics of the Ada source code submitted for compilation. Once all the necessary machine and compiler requirements are met (successful compilation), the object code is generated for the target machine. Since Ada does not require a particular operating system or executive, RTEs can exist on bare machines or on machines with an operating system or executive on board.

With a bare machine, all services not provided by the machine must be made available through the RTE for any applications programs. Obviously, if more Ada features are directly supported by the machine, a smaller RTE is required. Conversely, if little support of Ada features is present, then a much larger RTE will be needed. An obvious advantage to the bare machine approach is the portability of the code.

On a machine with an executive, the RTE can work cooperatively with the executive to support the Ada applications. Varying degrees of cooperation can exist based on the implementors decisions concerning the RTE capabilities. As

discussed above, support rendered by the operating system or executive will decrease the responsibilities of the RTE. This effectively reduces the size of the RTE. Reliance on the executive is not without drawbacks, however. The biggest drawback is that the generated object code will only run on systems that are targeted for the compilation system/machine combination.

This discussion of RTEs presented an overview of the RTE's development and current configurations. The discussion of the Ada RTE focused on the compilation system's responsibilities when used on either a bare machine or a machine with an on board executive. The difference between the Ada RTE and Ada RTS was also pointed out. This information is the basis for all other discussion of Ada RTEs and RTSs throughout this thesis.

Concurrency Revisited

In Chapter 1, a short discussion of concurrency spelled out a general notion of concurrent processing. Basically, it is the ability of a process to execute several statements at once. True concurrent processing can occur only when sub-processes are capable of executing regardless of the status of other peer sub-processes. As long as the peer processes do not share any information between them, they can run with no temporal considerations of one another. If they do share information, some scheme for ensuring non-corruption of the common information must be implemented.

In general, true concurrency can only be achieved on multi-processor systems. That is, the peer sub-processes are each assigned to one of the processors in the system or they are allocated across the available processors via some scheduling scheme. The end result is true concurrent processing.

In the single-processor system, however, things are not quite as nice. The CPU is shared by all sub-processes as needed, so n equal priority sub-processes on a single-processor machine only get 1/nth (n sub-processes) of the total CPU cycles. Furthermore, the sub-processes are either directly or indirectly affected by their peers. Each sub-process must wait for a peer sub-process to finish with the CPU in order take its turn. The time spent waiting is dead time for suspended sub-processes.

Many schemes exist for scheduling sub-processes into the CPU (Peterson and Silberschatz, 1985:Ch 4). Of particular importance to this work are the scheduling algorithms used in the Ada RTE to schedule tasks on a single-processor machine.

Concurrency Model for Ada. The model for concurrent processing in Ada is based on many of Hoare's desirable properties of concurrent programming facilities (Hoare, 1978). The following four properties of concurrent programming are considered very desirable for a satisfactory implementation of concurrency:

1) Security from Error: writing correct sequential code is hard enough. With the use of concurrency come many new problems associated with time-dependency errors.

2) Conceptual Simplicity: the structure of the concurrency facilities should promote sound development techniques and ease of understanding.

3) Efficiency: high-level language implementations of concurrency must address timing and storage constraints. If the high level language implementation is inefficient, then it will not be used.

4) Breadth of Application: the concurrency facilities must be suitable across a wide variety of applications. High-level capability for hardware interfaces and interrupts must be available so device drivers can be written easily (Hoare, 1978:666-677; Gehani, 1984:29).

The main idea in Hoare's model of concurrency is that the concurrent processes are sequential threads executing simultaneously. When it becomes necessary for the threads (sub-processes) to communicate, a rendezvous is made (via parameter passing) between the calling and called sub-processes. Once the exchange is complete (it should be as short as possible), the rendezvous is terminated. Processing then reverts back to parallel asynchronous sequential processing of the multiple threads within the program (Hoare, 1978:666-677).

Concurrency and the Single Processor Machine. The limitations of concurrent processing on the single-processor system stem from the fact that each sub-process cannot execute at will. If all sub-processes are to be considered peers with equal priority, then no true concurrent processing, in the rigorous sense, takes place. It can be argued however, fairly convincingly (Howatt, 1988), that if the speed of the single-processor system is fast enough and sub-processes are allotted CPU cycles fairly, the appearance of concurrent processing can be

achieved. Given this argument, the boundary between true concurrency and the pseudo-concurrency becomes blurred.

Real-Time Control and Task Scheduling in Ada. The preceding discussion of concurrency was given in general terms. The following discussion of real-time issues is given in the context of single-processor implementations of Ada tasking. That is to say, concurrency features are discussed as though they apply equally well to single or multi-processor machine architectures.

While Ada is a very good language in general, it does not appear to live up to some of its design goals. Particularly, the capabilities of Ada, regarding real-time embedded systems, fall short of the desired mark (Burns, 1987; Locke, 1987; McCormick, 1987). The Ada features used to support real-time processing are the delay statement and the pragma priority.

Currently, the implementation of the delay statement requires only that the delay be a specified minimum amount of time. The actual amount of delay provided by the RTS of a particular application may be significantly more than that requested by the programmer (DoD, 1983:9-11; Booch, 1987:288). This has obvious negative effects on the programmer's ability to enforce strict timing constraints using the delay statement.

Since real-time applications are generally characterized by some degree of predictability (Locke, 1987:51; McCormick, 1987:49), the inability of the delay statement to produce the exact delay requested suggests that the delay statement is inadequate for its designed intent.

Of equal concern to the real-time community is the manner in which individual tasks are scheduled for execution by the RTE. In a real-time system, it is desirable that the priority of any given task be dynamic. That is, as the system responds to the external world, the services provided by the real-time application should be quickly modifiable (Locke, 1987:53). For example, if a tactical fighter has several modes of operation, each of which is substantially different in terms of the real-time configuration of the on board computer, it is necessary that the RTS be capable of making adjustments based on the mode of operation.

The pragma priority in Ada does not allow the priority of a task to change dynamically. Once it is set to some level in the source code, it remains at that level. Task priorities cannot be changed during the course of program execution (DoD, 1983:9-17; Locke, 1987:53). The lack of this capability presents a problem for RTSs, because they need to be able to modify the system configuration based on external conditions.

Conditional Entry Calls in Ada

Section 9.7.2 of the LRM (DoD, 1983:14-15) spells out the operation of the conditional entry call. Other references (Elbert, 1986:422-423; Gehani, 1987:44; Shumate, 1988:50) also discuss the proper use of the conditional entry call.

Basically, the conditional entry call is a mechanism which provides a calling task with an alternative sequence of

executable statements if the called entry point is not immediately available. Immediate availability for rendezvous is discussed in the LRM (DoD, 1983:9-15, Para.4). The use of the conditional entry call for this research exploits nesting to set up an implied priority scheme. More details of this problem and the proposed solution are presented in Chapter 4. Very little information about the conditional entry call was found in the literature.

Use of Generics in Ada

Generics in Ada provide the capability to create templates of commonly used data structures for use in applications. The basic idea behind generics is to increase programmer productivity by using the templates for data structures that may appear many times in the same program. Good examples of candidates for generic usage are sort routines, incremental counters, and any other data structures that occur repetitively throughout general applications programs.

Some of the immediate consequences of using generics are clearer code, smaller amounts of source code, better reliability when using a proven generic and ease of modification. All these consequences of using generics generally enhance the programmers' productivity (Booch, 1987:243-244).

On a negative note, however, the use of generics can lead to some undesired results. Namely, if the code is written for use in a memory size sensitive application (avionics for example),

excessive object code generation by the Ada compilation system may preclude the use of generics. Since Ada was developed for use in conjunction with such systems, it is necessary to determine the effects of using generics in place of explicitly typed programs.

Types of Generic Units and Parameters. Generics are fairly well understood in terms of how they are used. Most textbooks on Ada devote a complete chapter to generics and their usage (Elbert, 1986; Booch, 1987).

There are two basic units that can be created in the generic format: subprograms and packages. Formal parameters used in generic units fall into one of three categories: object parameters, type parameters, and subprogram parameters. These parameters are used in much the same way as normal parameters in Ada.

It is also possible to have no formal parameters for a generic unit. This just means all information about the generic unit is already available in the generic specification.

A generic unit in the source code of a particular application must have a corresponding instantiation to be of any use. The instantiation of the generic notifies the compilation system that the programmer wants to use the generic at the specified place in the code. Once this is done, the generic performs as if it were hard coded at the point of instantiation (Elbert, 1986:268).

Problems with Generics. As mentioned above, generics do have slight drawbacks in memory sensitive applications. Since the object code resides in memory, it should be as small as possible in these applications.

As mentioned in Chapter 1, there is a general consensus among users that virtually every Ada compiler produces more object code when generics are used in place of explicit routines (King, 1988; Lyon, 1988). Several reports also indicate that the use of generics will cause the object code size to expand (Softech, 1986; ARINC, 1987).

The current solution to the problem of object code expansion in code which uses generics is to write the code first without regard to efficiency. Once it is correctly implemented, the inefficient code segments are optimized as necessary to bring the object code size down to acceptable limits (Softech, 1986:3-9).

This discussion of the use of generics in Ada was used to point out some of their structural features and limitations. Of critical importance to the real-time users of Ada is the expansion of the object code when generics are used. This issue is explored in more depth in Chapter 5.

Software Testing Methods

Compilers are programs written to automate the process of communicating with the underlying machine architecture. Since they are software programs, common practice dictates that they be tested for proper performance. Software testing is professed to

be quite an important step in the development life cycle for software products (Jensen, 1979:38; Pressman, 1987:467). A successful integration of verification and validation throughout the software development life cycle may be the difference between overall project success or failure (Woffinden, 1987). Hence, the only prudent approach to software testing is to use it throughout the development cycle.

Many testing techniques are practiced by software developers. The particular test methods used depend upon the type of project, the resources available, and the management philosophy regarding testing. While testing is a very important part of development, it is often undertaken on the back end of the project. As such, time is generally at a premium, and the test team may only be able to perform cursory functional testing of the software.

Testing general purpose applications software is a fairly straightforward process. The reader is referred to Jensen (Jensen, 1979:Ch 5), Howden (Howden, 1987) or Pressman (Pressman, 1987:Ch 14) for detailed explanations of verification and validation techniques.

Compiler Testing Methods. Compilers are special purpose computer programs. They materialize from the best efforts of the developers to represent the syntax and semantic structure of a particular language.

In general, most large programs will contain some errors (Jensen, 1979:329; Pressman, 1987:468-469). Since compilers are

special purpose applications, it can be assumed that they, too, will contain errors. As this is the case, testing methods to uncover errors in compilers should follow accepted techniques as described in the references mentioned above.

A complementary testing technique for testing compilers is to perform regression testing as the compiler evolves (Aho, 1986:731-732). Regression testing can be viewed as an iterative process of test case development. As the compiler matures (incorporates new features), new tests are written to exercise the newly added features. These new tests, plus all the old tests, are successively run against the latest version of the compiler. In this manner, the developer can check the consistency of the compiler operation with that of its predecessor. If, after some features are added (assuming no changes to the original version except add-ons), the compiler reacts incorrectly to some of the older tests, then the developer knows that the current modifications are adversely affecting the overall operation.

Ada Compiler Validation Capability. Regression testing provides good consistency of the product through its development. A good case in point is the use of the Ada Compiler Validation Capability (ACVC) to validate Ada compilers for use in DoD.

Although use of the ACVC does not strictly constitute regression testing, the ACVC test suite is constantly growing to reflect more and more testing coverage. As the number of tests in the ACVC increases, each compiler submitted for validation is

tested more and more thoroughly. Thus, a kind of regression testing is occurring by virtue of the dynamic nature of the test suite itself.

The approach used to validate Ada compilers in DoD is outlined by Goodenough (Goodenough, 1981:57-64). The emphasis in the ACVC is on development of many small tests to ensure that each of the Ada language features is implemented correctly. One difficulty with this approach (attributable to Ada's size) is the shear magnitude of the number of tests. To date the ACVC test suite contains over 3500 individual tests (Wilson, 1989).

Another difficulty, with using many small tests, is that the effects of complex combinations of Ada features cannot be tested. In 1984, McDonnell Douglas Astronautics Company began work on the Common Ada Missile Packages (CAMP). CAMP was a research project directed at determining the viability of reusing Ada parts (generics) in real-time embedded systems. Initially, many problems with validated compilers caused slow progress in the work. It was not until 18 months after the start of the project that any of the validated compilers were able to handle the complex Ada code generated during the research. Understandably, the response of the researchers to the ACVC was somewhat mixed. They acknowledged that the ACVC efforts were commendable in establishing a baseline standard for Ada compilers. They added, however, that the test suite could not meaningfully test many of the complex programs written as a part of their research. They went on to suggest that the approach used to develop tests for

the ACVC be expanded to include tests of complex Ada code, particularly in the area of generic usage (Herr and others, 1988:75-86).

In light of the comments by researchers at McDonnell Douglas, some adjustments to the testing approach used in the ACVC are needed. It appears that the ACVC tests lean toward testing for an absolutely correct implementation of the language. It does not test the possible complexities inherent in the Ada language, particularly in the area of generic usage.

Summary

The literature reviewed for this thesis is by no means all encompassing. Much more information about these areas exists. This review provides a good baseline of information about the proposed areas of study. It also provides a good starting point for similar work in this vein.

III. Testing Ada Run-Time Task Scheduling

Introduction

As mentioned in Chapter 1, the main thrust of this chapter is to develop and use tests which reveal the degree of fairness of Ada task scheduling within the run-time environment on a single-processor machine. Once the tests are developed and data are gathered using available compilers and machines, statistical tests will be used to assess whether a given RTS has a fair task scheduling algorithm.

Preliminary Design

The approach used in solving this problem consists of constructing small test programs to generate utput that gives some insight into the nature of the underlying scheduling algorithm. First, a series of n, equal priority tasks (Figure 3) were used to determine how the RTE scheduler services autonomous tasks. Each of the tasks, T1..Tn, ran for the duration of the test with no interruption from peers.

Next, as shown in Figure 4, n interdependent equal priority, tasks were run to generate data on how they were scheduled by the RTE. Tasks T2..Tn all called separate entry points in T1. The Ada RTE uses a First-In-First-Out (FIFO) scheduling scheme for servicing the calls queued on each entry point (DoD, 1983:9-9). This scheduling scheme should produce a fairly uniform distribution on the number of times each calling task rendezvous.

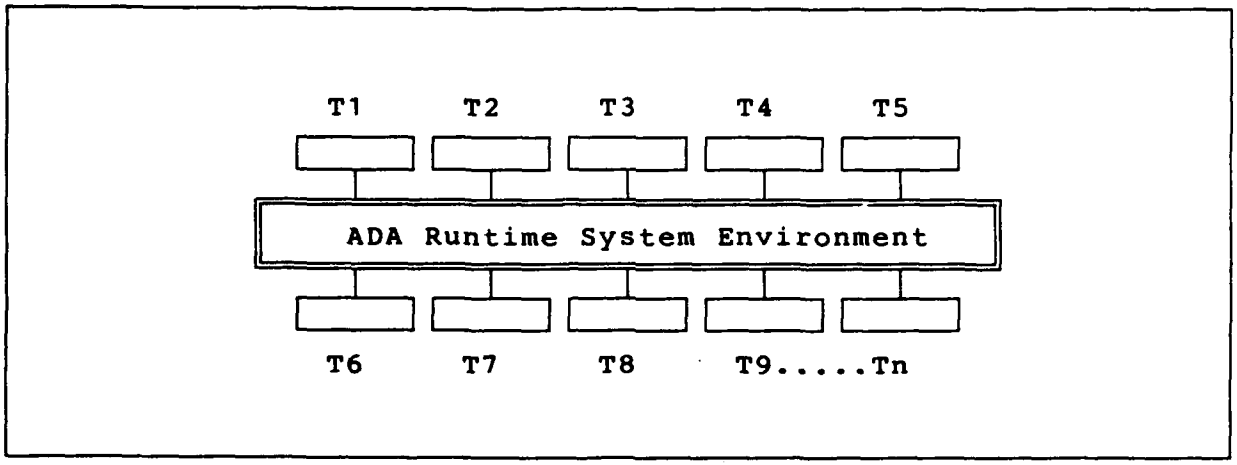

Figure 3 Ada Run-Time System for Test 1

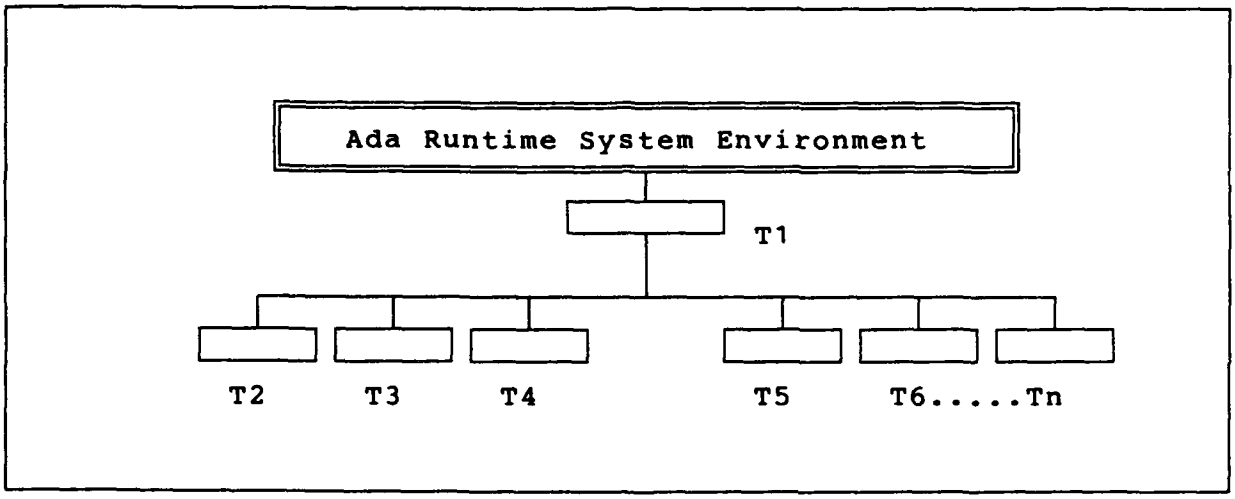

Figure 4 Ada Run-Time System for Test 2

In each of the above test cases the tasks were identical. Integer counters within each task body were incremented each time the task was executed. This provided insight about the amount of processor time allocated to each task.

Finally, a combination of the first two tests was run to generate data on the hybrid case (Figure 5). The intent here was to determine the effects of the different task scheduling requirements on the run-time scheduler. Intuitively, a completely fair algorithm would allow an equal number of task executions across all tasks on the system. This last test showed whether this is the case for a given RTE. Any substantial departure from a uniform distribution across all tasks might be traceable to the overhead required for handling the rendezvous.

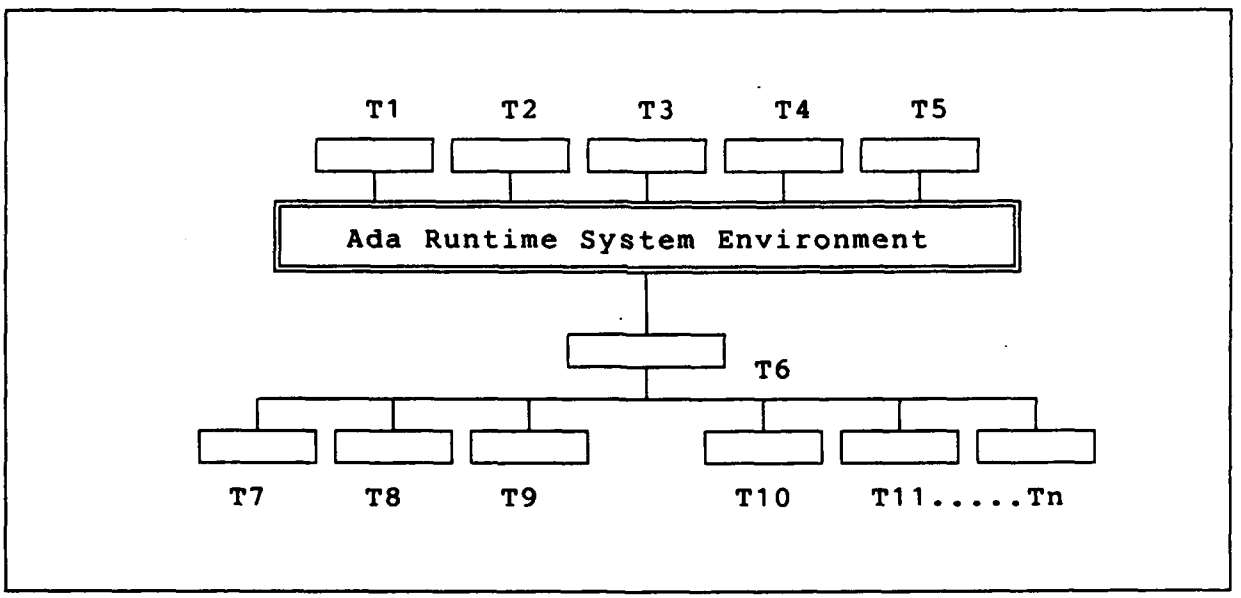

Figure 5 Ada Run-Time System for Test 3

In general, if the integer counters showed a uniform distribution across all tasks in a particular test, then it could be inferred that the scheduling algorithms used were fair. The fairness of each compiler was tested using data generated by

running the test programs on a specific compiler/machine combinations.

Detailed Design

The implementation of the three tests described above was fairly straight forward in Ada. Since Ada is fairly well defined by the LRM (DoD, 1983), the overall structure of the tests were determined from the preliminary design discussion. Each of the three cases was derived from the conceptual view of the structure presented by the figures above.

Test 1 Design. The idea behind Test 1 was to expose the nature of the scheduling algorithm when tasks were executing, but not calling other tasks. This test used autonomous tasks which did nothing more than sit and spin. Each autonomous task incremented an integer counter in its body. Figure 6 shows the simple task body of one of the tasks in Test 1.

The idea here is to get all the tasks up and running, but not start any of them counting until they are all active. The WHILE loop in the task body facilitates this action. The flag TASK_ACTIVE is set to TRUE in the main program after all tasks are activated. This structure initially restricts all processing within the tasks to execution of the null statements. When all n tasks are ready, the flag is set and each one begins to increment the appropriate counter. Thus, each task should have an even chance at incrementing its counter during the duration of the test (assuming of course the scheduler is fair).

The delay statements seen in the loops of the tasks became necessary because most of the compilers went into infinite looping without them. The purpose of the delay here is to allow the RTS an opportunity to break from a loop and continue processing. Without the delay statements in the loops, the RTS simply allowed processing to continue in one of the running tasks. Hence, no other processing was accomplished, and no data was generated. Adding the delays created a break point from the infinite loops and allowed the test programs to run to completion.

```
task body NUMBER_1 is
    begin
        while TASK_ACTIVE = FALSE loop
            delay DURATION'SMALL;
        end loop;
        loop
            delay DURATION'SMALL;
            COUNTER_1 := COUNTER_1 + 1;
            exit when TASK_ACTIVE := FALSE;
        end loop;
    end NUMBER_1;
```

Figure 6 Task Body for a Task in TEST 1

Test 2 Design. The crucial difference between Test 1 and Test 2 was that, in Test 2, the effects of handling the rendezvous came into play. Since there was some non-zero amount of time involved in establishing the rendezvous, the Ada RTS must compensate in the scheduling of the waiting tasks. Again, intuitively, the fair scheduling algorithm should provide equal time to equal priority tasks.

Figure 7 shows the internal structure of the called task for Test 2. The structure of this called task is such that the calling tasks access their own entry point. A running count of the number of times the siblings rendezvous is maintained as COUNTER_6.

The siblings (Figure 8) also track their individual counts so they may be compared after execution of the test program. Since the scheduling of the calling tasks within the select statement is arbitrary (LRM, 1983:9.7.1) and all tasks have equal priorities, any distribution other than uniform would be a surprise.

```
loop
     select
          accept INCREMENTER_7 do
               COUNTER_6 := COUNTER_6 + 1;
          end INCREMENTER_7;
     or
          accept INCREMENTER_8 do
               COUNTER_6 := COUNTER_6 + 1;
          end INCREMENTER_8;
                    .
                    .
                    .
     or
          accept INCREMENTER_11 do
               COUNTER_6 := COUNTER_6 + 1;
          end INCREMENTER_11;
     end select;
end loop;
```

Figure 7 Internal Structure of T6 in Test 2

Test 3 Design. Test 3 is the hybrid case of combining Tests 1 and 2. In other words, Test 1 tested autonomous tasks running freely on the run-time system and Test 2 introduced the effects of the rendezvous on the scheduling algorithm. Now, if these two cases are combined, then the results will show any influence of one on the other.

```
task body NUMBER_7 is
begin
   while TASK_ACTIVE = FALSE
     loop
         delay DURATION'SMALL;
     end loop;
     loop
         delay DURATION'SMALL;
         COUNTER_7 := COUNTER_7 + 1;
         NUMBER_6.INCREMENTER_7;
         exit when TASK_ACTIVE = FALSE;
     end loop;
end NUMBER_7;
```

Figure 8 Structure of Sibling Task in Test 2

The design of this test case simply required combining Tests 1 and 2. Since there is no new information needed regarding the structure of Test 3, discussion of it is not necessary. Figure 5, Ada Run-Time System for Test 3, provides all the conceptual information needed to construct the test.

Expected Results. Prior to performing any analysis on data generated by these test, some thought was given to expected results in the context of what is known. The underlying

assumption for all of these expected results is that the RTS provides each task an equal amount of CPU time.

The first test should show a uniform distribution over the counters because each task is autonomous. The only influence on the individual tasks is that of the run-time scheduler. Since the tasks all have equal priorities, anything other than a uniform distribution would be cause for concern. The scheduling order of tasks with equal priority is not defined in the LRM (DoD, 1980:9-16). However, if the distribution of the task servicing is not uniform, then the scheduling order invalidates the equal priority constraint, by definition.

The second test should also generate a uniform distribution, because of the arbitrary nature of selection within the select statement. Hence, on the single processor machine, some sort of non-preemptive, equal priority scheduling scheme should be evident.

The expected results of the third test are hard to determine. If based entirely on the equal priority assumption, then the expected results should show a good uniform distribution. However, with the necessity to schedule equal priority tasks based on two different relationships to the run-time environment, the expected results are non-deterministic. If the overhead for establishing the rendezvous is substantially greater than that for switching between autonomous tasks, then there is a good chance the distribution of counter values in Test 3 will not be uniform. Additionally, since there is no

rendezvous overhead for the autonomous tasks, all of them are always ready for execution. This could also cause a skewing in the distribution of counter values.

Statistical Methods to Test Fairness

The statistical method used to test the fairness of the scheduling algorithms is the Chi-Square Goodness-of-Fit Test. The hypothesis to be tested is as follows:

H_0: Underlying Population Distribution is uniform or

H_A: Underlying Population Distribution is other than uniform.

The significance level, α, is to be taken at .05. This value of α is sufficiently high to provide good confidence if the null hypothesis is rejected by the Goodness-of-Fit Test.

Each separate test was run 30 times to generate a pool of data from which observed values for each of the cells were calculated. The test sample was then used to test the hypothesis stated above. Sample calculations on data taken from one of the compilers are included in Appendix A. Additionally, all data from all tested compilers are shown.

Table 1 shows results from all compiler/machine combinations. The information in the columns under the heading TEST 1, TEST 2, and TEST 3 give the results of the statistical testing. YES means the null hypothesis was rejected and NO means it was not.

Table I Test Results for Part 1

MACHINE	COMPILER	OS	TEST 1	TEST 2	TEST 3
Elexsi 6400 (ICC)	Verdix Ada 5.5	UNIX BSD 4.3	NO	NO	NO
VAX8650 (ASD/SCEL)	DEC Ada Ver 1.5	VMS Ver 4.7	NO	NO	YES
VAX11/780 (ISL)	DEC Ada Ver 1.4	VMS Ver 4.6	NO	NO	YES
VAX11/785 (SSC)	Verdix Ada 5.41	UNIX BSD 4.3	NO	NO	YES
Z-248	JANUS Ada Ver 2.02	MS-DOS Ver 3.2	NO	YES	YES

Implementation Problems

Although the algorithms developed for use in this problem were fairly straightforward, some of the resulting problems were not anticipated. The most worrisome of the problems encountered was a problem of the RTS's inability to manage CPU time among the tasks. When the RTS started the first task, it never stopped it to start another one. This was the case on the majority of the machines. The ELEXSI 6400 could handle the code but the scheduling of tasks was severely skewed in favor of the autonomous tasks. The autonomous tasks were executed thousands more times than those with embedded calls. Since rendezvous severely degraded the fairness of the scheduling algorithms in the RTS, a work around to this problem was implemented. As mentioned earlier, delay statements were added to the task bodies

to interrupt the task and allow the RTS to perform its scheduling function.

Results. The overall conclusion to be drawn from this portion of the research is that task scheduling is reasonably fair when comparing like circumstances. In other words, if all the tasks are autonomous, then the scheduling is fair. If all the tasks rendezvous, then the scheduling algorithms perform marginally well. But, if the hybrid case of task structures is used, then most of the compilation systems fail to provide fair scheduling in the strict sense.

It may be unreasonable to expect a RTS to schedule an autonomous task and a task with rendezvous fairly, given the overhead of the rendezvous. The results of these tests indicate that, in most cases, it would not be a good assumption to presume equal priority tasks of differing structures (autonomous or dependent) will be scheduled fairly.

The complete test data from several available compiler/machine/operating system combinations are contained in Appendix A. Further conclusions and recommendations for this portion of the research are contained in Chapter 6.

IV. Conditional Entry Calls and Implied Priorities

Conditional Entry Calls

As discussed in Chapters 1 & 2, the second area of interest in this thesis is the conditional entry call. The complete specification, syntax and semantic usage for this feature is contained in the Ada LRM (DOD, 1983:Ch 9, 14-15). An example of its use is given in the LRM to assist the reader in understanding the basic operation of the feature. Further, and slightly more illuminating examples, are available in Shumate's book (Shumate, 1988:121,520-531) and Elbert's book (Elbert, 1986:420-425).

From the standpoint of the Ada programmer, the conditional entry call is a very useful feature when writing code for real-time concurrent processing. When a call is made to a conditional entry in a task, the Ada run-time environment responds by either completing the rendezvous, if rendezvous is immediately available, or by executing the "conditional" response if rendezvous is not successful.

This conditional response is a single alternative, in the form of an "else" option, which is executed in place of the unsuccessful rendezvous attempt. The conditional entry call allows a task to continue processing when the called task is busy, instead of suspending it until the called task can rendezvous.

A suspended wait can harm a system's performance in a real-time, embedded environment. If time critical portions of the

software system depend on the task, which may not be available because of suspended waiting, the real-time performance of the overall system will be degraded. Thus, the avoidance of suspended waiting enhances the overall system's capability to perform in a real-time environment.

Example Use of Conditional Entry Call. As an example of the use of the conditional entry call, Figure 9 shows a procedure written in Ada that performs busy waiting very nicely (Elbert, 1986:423). This procedure prevents the calling task from being suspended on a call to SEMAPHORE.WAIT. If the entry point SEMAPHORE.WAIT is busy when called, the else path is selected and the task repears the select statement until the rendezvous is made. Conditional entry calls can be used for more than just busy waiting. For instance, a calling task can return to execution and try the entry point at some later time.

In the example of Figure 9, the call to entry point SEMAPHORE.WAIT will be made only if rendezvous is immediately available. Otherwise, the "null" statement in the else branch will be executed. Since the only way out of the loop is for the return statement to be executed, this structure provides busy waiting for as long as the entry point into the called task is busy.

```
procedure Admit is
   begin
      loop
         select
            SEMAPHORE.WAIT;
            return;
         else
            null;
         end select;
      end loop;
   end ADMIT;
```

Figure 9 Pseudo Code Representation for Busy Waiting

Nested Conditional Entry Calls

An issue of particular interest to the Ada Validation Facility regards the effects of using several levels of nested conditional entry calls to imply a priority scheme over a set of entry calls. In other words, can a programmer construct Ada code, using the conditional entry call, that will provide a priority scheme within the scope of the nesting and will the code work as expected? The balance of this chapter will discuss my resolution of this question and present a summary of the results of testing the coded solution across several available compilers.

Preliminary Design

Of major importance in the design solution to this particular problem is the functionality of the overall construct and not the details of "what it can do in real terms". In other words, the intent here is to show only that the algorithm discussed above can be implemented to work correctly.

The solution to the problem will exhibit the following characteristics:

1. Process entry calls based on ready queues of the entry points.
2. Calls to the highest level will be processed until they are exhausted. Then the next level down will be processed.
3. Execution will stop only when no more calls are queued at any entry point.

Detailed Design

The form of the algorithm used to solve this problem is determined by the problem statement. That is, nested conditional entry calls are used to create an implied priority scheme over the scope of the nesting. The particular solution here calls for the else options of each successively lower conditional entry call to contain the next level select statement.

The use of the conditional entry call implies the use of tasks. The code structure needed involves a controller task which contains the entry points for the different level priorities. Other tasks in some quantity, n, call the controller task entry points to gain access to a particular priority level

operation. Figure 10 shows the general structure of the solution in block diagram form. The entry points within the controller task can be accessed using calls to HIGHEST, MIDDLE and LOWEST.

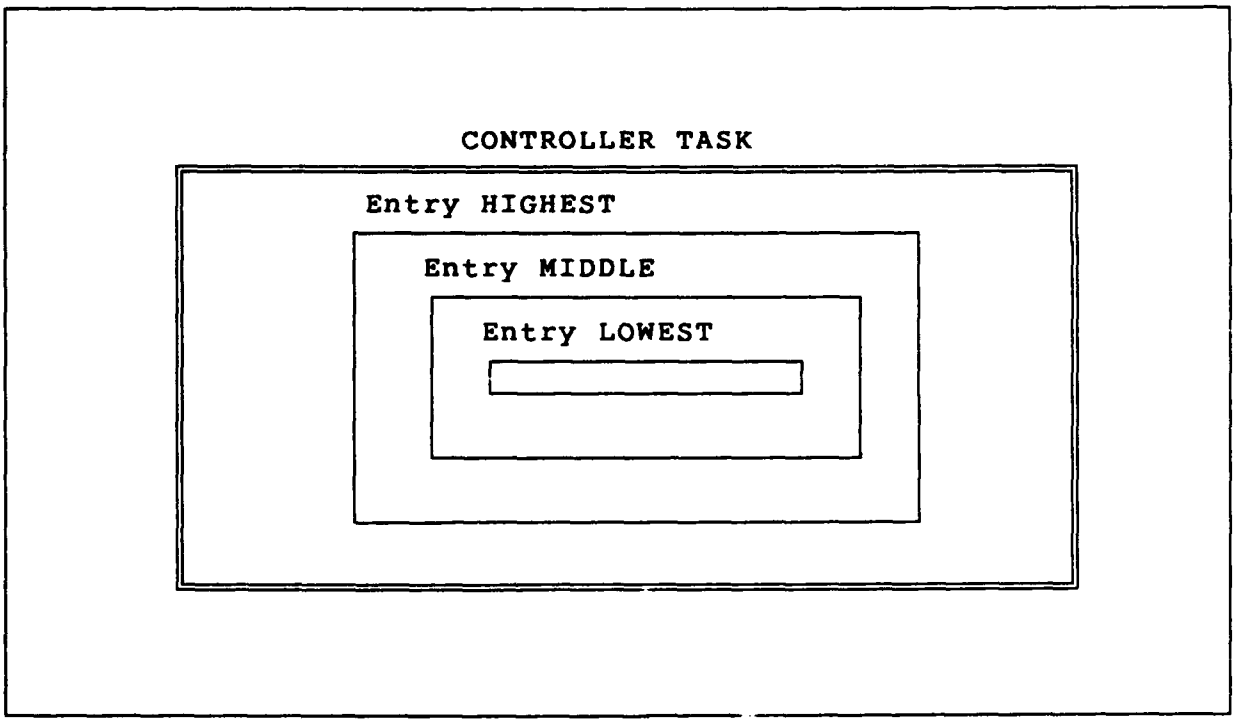

Figure 10 Block Diagram of Nesting Solution

Accessing particular priority levels correctly is sufficient evidence that the algorithm works. The nested conditional entry calls will be in a main loop and the ready queues for each entry point will determine whether execution occurs.

When no more calls are queued at any entry points, the alternative statements at the deepest level conditional entry call determine what will occur next. Available options could include delaying until more calls queue up, exiting the loop, or even terminating the task. The test written to demonstrate the

even terminating the task. The test written to demonstrate the solution uses the middle option, exiting the loop.

Implementation

The first step in implementing the solution was to decide the coarseness of the available priority scheme. Since the availability of a particular entry call within the controller task is two-valued, either available for rendezvous or not available for rendezvous, the total number of possible combinations of entry points in the controller task is 2^n, where n is the number of entry points.

Additionally, the combinations of the binary representations of entry points determines the number of test cases needed to test the solution. For this solution, three entry points were used. This meant a total of eight test procedures were needed to exercise all combinations of entry point calls. The calling tasks in each test were designed to call a CONTROLLER entry point and nothing more. The CONTROLLER then prints a message telling what priority levl call was processed. Since the scheduling of accepts in a select statement is arbitrary (LRM, 1983:9.7.1), proper ordering of the output results sufficiently demonstrated that the algorithm performed correctly.

Table II shows the various combinations of calls to CONTROLLER by the other tasks in each program. Two extra combinations were tested to determine the results of multiple calls to a single level. An X in a column means a call was made

at that level. An 0 means no call was made at that level. The structure of the calling tasks were identical and consisted only of calls to entry points within CONTROLLER.

Table II Calling Task Combinations to CONTROLLER

	Calls To CONROLLER Task		
Test #	HIGHEST	MIDDLE	LOWEST
1	X	X	X
2	X	X	0
3	X	0	X
4	X	0	0
5	0	X	X
6	0	X	0
7	0	0	X
8	0	0	0
9	X X	0	X
10	X	0	X X

The approach, as mentioned above, was to create a task type CONTROLLER to be used for each test procedure. Figure 11 shows the specification of the CONTROLLER task type. The three entry points shown in Figure 11 are called by other tasks in the test program according to Table II. The algorithm is constructed so that calls to CONTROLLER.HIGHEST are answered until there are no more calls for that entry point.

```
task type CONTROLLER is
   entry HIGHEST;
   entry MIDDLE;
   entry LOWEST;
end CONTROLLER;
```

Figure 11 Task Type CONTROLLER Specification

The task body of CONTROLLER is depicted in Figure 12 and shows the general structure of the solution. Pseudo-code representation of the solution shows the loop structure containing the nested conditional entry calls. Here, the exit in the terminal "else" option was used to end processing when no more calls were pending.

The executable solution (each test procedure has an instance of task type CONTROLLER) simply displays the status of the current process with strategically placed Ada PUT statements. These PUT statements allow the user to follow the progression of the test procedure in near real-time from the terminal.

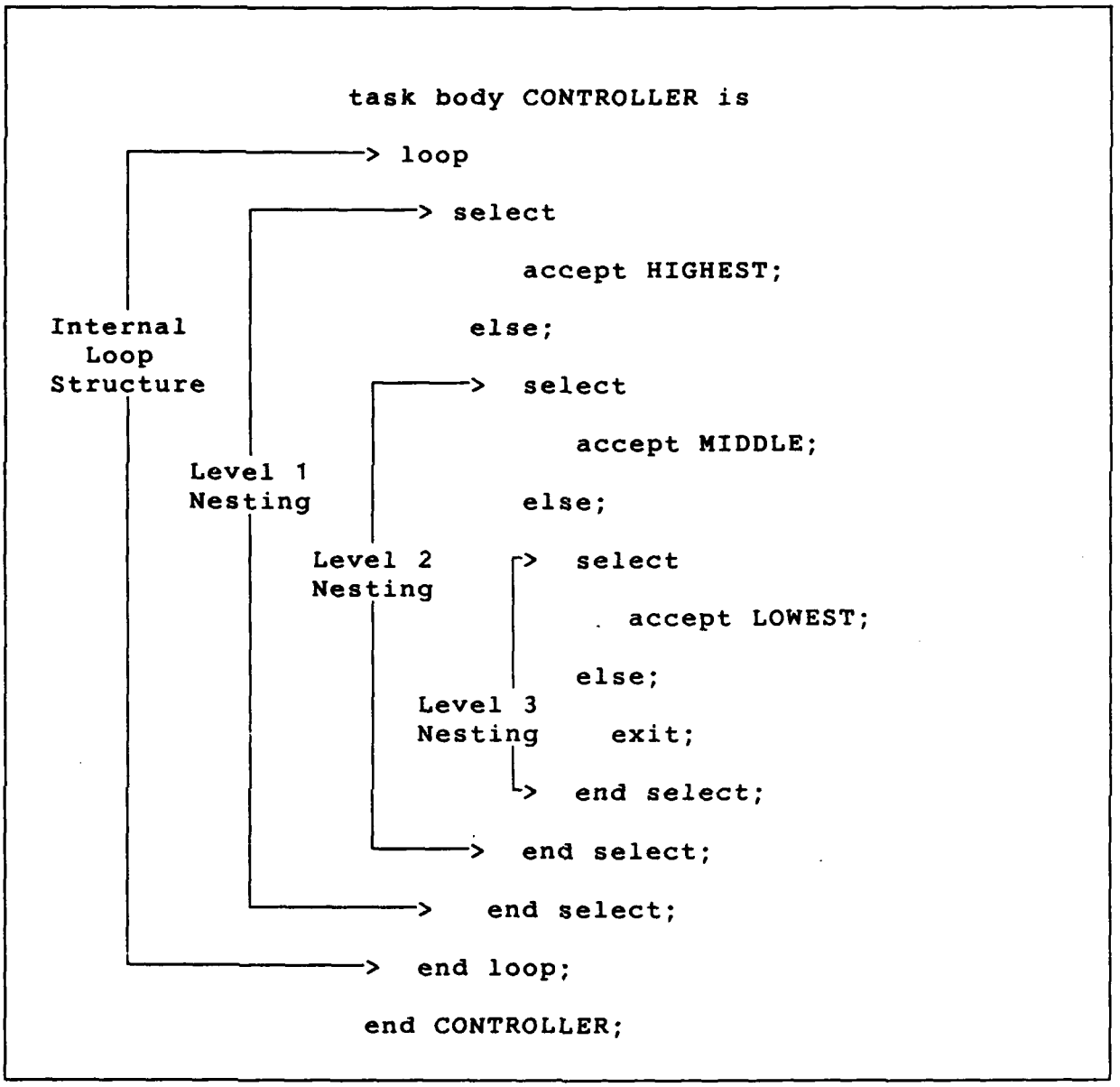

Figure 12 Pseudo Code Representation of CONTROLLER TASK

Results

Appendix B shows the results of running the test program. Every compiler/machine/operating system combination used in this section of the research produced the same results. A list of

these is also included in Appendix B. The implementation of this solution performed correctly on every system combination tested. The results show conclusively that nested conditional entry calls can be used to enforce an implied priority scheme.

V. The Effects of Using Ada Generics on Object Code Generation

Introduction

The main purpose of this chapter is to discuss the design and implementation of a suite of test programs developed to determine the effects of using Ada generic constructs on the generation of object code. It is commonly accepted (Lyon, 1988; King, 1988) that using generics causes some inflation in object code size. This portion of the research will develop a test suite that will show whether this common notion is indeed true.

The preliminary and detailed designs will be discussed in the following sections. This discussion will outline the steps taken to develop the test suite as well as explain the content of the test programs. All results from the tests are consolidated in Appendix C for easy reference. Finally, problems encountered during the implementation phase and summary conclusions are discussed in the last sections.

Preliminary Design

The problem statement in Chapter 1 stated the need to determine the effects of using generics on the generation of object code. Obviously, to test the effects of using generics, both generic and explicit versions of the test programs should exist. The generic version and the explicit baseline version should perform identically and be very similar in structure.

Gehani (Gehani, 1987: 215) points out that explicit and generic versions of Ada subprograms and packages differ only in their specifications. Thus, translating an explicit version of a program to its corresponding generic form should only require altering the specification of the explicit version.

In general, then, the approach used to select the test code was to choose existing solutions to common data structures implementations. Code representing these various implementations are widely available in the literature. Of particular interest for this part of the research are those code segments containing one or more of the types of generic formal parameters.

Detailed Design

There are several types of generic formal parameters. Each of these parameter types can be used alone or in combination in generics programs. Also, a generic program may have no parameters. Therefore, the test programs in this test suite cover each case of parameter usage. The reader is referred to Elbert (Elbert, 1986: Ch 5) for an excellent discussion of the generic formal parameter types.

Generic formal object, type, and subprogram parameters are used in programs in the test suite. Also, included is a program with no generic formal parameters. Thus, the test suite covers the range of possible generic formal parameters.

Selection of Test Code. The selection of code to use in the test suite was fairly simple. During the literature review for

this problem, many texts were found that examined the generic facility of Ada (Elbert, 1986; Booch, 1987; Gehani, 1987; Shumate, 1988). The programs in the test suite come from some of the texts used for this review. Each of the programs chosen had some attribute which made it useful for this study.

In general, the generic form of the program existed and the explicit version had to be written. While this goes counter to the preliminary design discussion, it proved to be more productive to first find the generic version, then code the explicit version.

Using this method for gathering test programs, six code segments with generics were selected. Each of the generics used can be seen in Table 2 with its corresponding generic formal parameter type list.

Table III Generic Test Programs and Use

Program Name	Generic Formal Parameter	Usage
GENERIC1.A	None	COUNTERS
GENERIC2.A	Object and Type	STACKS
GENERIC3.A	Type	EXCHANGE
GENERIC4.A	Type w/package	QUEUE
GENERIC5.A	Type w/package	SET OPS
GENERIC6.A	Type/Subprogram	SORT

Implementation. Once the code segments were chosen, the explicit versions needed to be coded. Although the explicit versions were written using the algorithms from the generic versions, the translation was not a one to one mapping. If, for example, the generic version contained formal parameters, those parameters must be uniquely defined in the explicit version. A generic formal type parameter can be defined for a wide range of types, but the same usage in the explicit version requires that the type used be uniquely determined.

When all the translations were completed, the test suites were run on several available compilers (See Appendix C). Problems during the implementation phase are discussed in the following section.

Implementation Problems

Most of the problems encountered during the implementation of these tests centered around the use of generics. Only simple segments of code were used to cut down on the complexity of implementing the solution. Even with simple code, the confusion of using the generic formal parameters types was a major hurdle.

One major problem dealt with using a subprogram as a parameter. With the JANUS Ada compiler, there was no default value on the use of visible functions. Unless the programmer explicitly instructed the compiler to use the visible subprogram (insert the box <> notation after the subprogram specification),

it looked for an overloaded operator with the same name. If it was not found, an error was raised which indicated the subprogram parameter could not be found.

This problem did not occur on the mainframe compilers. Since the problem was unique to the PC compiler, the code was modified to work on there. Then the same version was used on the mainframes.

Results

As mentioned above, there is a notion among the users of Ada that generics generate more object code. While this notion is somewhat true, it is not completely true. In some cases, the generic version does generate more object code. In other cases, the opposite is true. In still others, both versions of the code produce the same amount of object code.

Clearly, then, several different factors go into properly answering questions about the generation of object code from generics. How was the compiler designed to handle generics? Does it insert the equivalent object code directly into the object module or is a copy of the module maintained and inserted as needed at run-time? Does the complexity of the generic cause any inflation with regard to the final object code size? Do the number and type of generic formal parameters affect the object code generated?

These types of questions must be answered in the context of a given compiler and machine architecture. It does not suffice

to say that generics cause inflation of object code size. More correctly, the object code size varies depending upon the particular Ada code, Ada compiler and machine used. It may be the case that there is a relationship between the object code sizes of the different versions. The object code sizes may be inversely related as the number of instantiations and copies increase.

As mentioned above, Appendix C contains all pertinent information concerning the results of these tests.

VI. Conclusions and Recommendations

The conclusions and recommendations in this chapter are derived from the results of this thesis. The conclusions are based on the evidence gathered during the research and the recommendations stem from unresolved issues/ideas encountered as the research progressed.

Summary Conclusions

Each sub-section addresses one of the three problems studied during the course of this research.

Part 1 Conclusions. This part of the research effort centered on testing the fairness of task scheduling algorithms in various Ada run-time systems. Tests were generated to highlight the effects of a variety of tasking structures in all programs, it was assumed all tasks had equal priority.

Using the Chi-Squared Goodness of Fit Test on the data generated from these programs, consistent patterns of results occurred. The data revealed that all the compilers (except Verdix 5.5 on the ELXSI 6400) failed to provide fair scheduling when the test program included both autonomous tasks and tasks with rendezvous. In a strictly statistical sense, the null hypothesis, that the scheduling algorithms were fair, was rejected whenever the Ada RTS was faced with the hybrid test case. This should be no real surprise, since the RTS overhead

for tasks containing rendezvous is substantially greater than for autonomous ones.

The data from these tests seem to indicate that the tasks closer to the top, in the select statement, are selected more often than those further away. This then, gives some indication that the selection scheme is not arbitrary, as defined in the LRM.

The compilation systems did much better when the tests were run using one of the other general test cases. Table I in Chapter 3 summarized the results and showed that four of the five compilers used were fair when using autonmous tasks exclusively. In the case of the tasks with rendezvous, three of the five were found to be fair.

It is interesting to note that by running the test code on the ENCORE with Verdix Concurrent Ada Beta Version (B1.1), the RTS only allowed rendezvous for two of the five accepts within a select statement. This was verified by switching the order of the accepts within the select statement and observing the output. This particular machine/compiler combination always chose only the first two entries after the select statement for rendezvous. This, then, hints at a problem with the compiler, since all open select alternatives should get some CPU time.

My overall impression of the Ada compilers used during this research, in regard to task scheduling, was that the majority of them still need work to insure that the fairness issue is resolved. While it may not be possible to get complete fairness

in task scheduling, I think there is sufficient evidence here to suggest more work can be done. I also believe that upcoming revisions to the LRM should include more specific details about these Chapter 9 implementation details.

Part 2 Conclusions. This part of the research was designed to show that an implied priority scheme could be implemented using the conditional entry call. It was shown that by nesting conditional entry calls within select statements, an implied priority scheme could be established.

Three levels of priority were established within the test program. Then, tasks from within the test program called the entry points at the different priority levels. Verification of the program was done with textual output statements. These statements traced the progress of the test program as it cycled through various combinations of calls to the entries.

All results were as expected. Conditional entry calls nested in select statements can generate the desired effect of an implied priority scheme.

Part 3 Conclusions. By far the most challenging portion of this research was Problem 3. This section of the research concentrated on determining if there was a difference in object code size caused by using generic program structures.

Test programs in this portion of the work included generics and explicit versions of those generics. Each was compiled and loaded to produce executable versions. Each generic executable was compared with the explicit version with respect to object

code size. It was determined that generics do not always produce larger object programs. Some explicit versions of code caused the generation of more object code than the corresponding generic. Thus, the only conclusion to draw from this research is that it is not categorically true that generics always cause object code inflation. This refutes the commonly held notion that generics do indeed inflate object code.

Recommendations

These recommendations come from thoughts about Ada and some of the interesting possibilities for continued research. Ada is continuing to mature and may some day actually become the language it was designed to be.

Language Recommendations. The current version of the LRM is intentionally vague on many implementation details. While some flexibility is a must in language specifications, I think defining the basic characteristics of language features is crucial. Chapter 9 of the LRM (DOD, 1983:9.7.1,9.8) does not define any restrictions on how the language should be implemented to provide several particular features. In each instance, the actual results are nondeterministic. The user is required to write test code to uncover the actual performance characteristics of a given compiler.

If the features were sufficiently defined and the compiler validation suite contained tests which exercised the features, the user could be more confident in the actual results.

Otherwise, the burden of proof of correctness shifts to the user, who may have little or no understanding of how to test his code for undocumented features.

I recommend that sufficient detail about a particular language feature be included in the LRM. There are many specifics already in the LRM, but there are also many that have been omitted.

Further Research Recommendations. In the course of this research many issues were raised. Some of the more interesting are offered here as future research areas.

Before mentioning any of the areas for future research I want to caution prospective researcher concerning research involving many different systems and/or compilers. My experience was that an inordinate amount of time had to be spent porting the test cases to each different machine/compiler combination and learning how each worked. Hence, my first recommendation to any researcher is to try to scope your work to one machine/compiler/operating system combination. This may not provide a wide breadth of information, but the depth of understanding of the single system will probably serve you better in the long run.

Recommendations for future research in the area of task scheduling deal with developing more sophisticated methods for discerning the actual algorithms used. A researcher might, for example, take a particular machine/compiler/operating system combination and concentrate on determining how the RTS schedules

different combinations of different types of tasks. While the compiler documentation should contain this information, many times the burden of discovery is left to the user. Undocumented features could also be noted. The point here is to find out as much as possible about how the compiler and RTS handle the overall chore of task scheduling.

Future research using the implied priority scheme algorithm developed in the second part of this research might enhance the algorithm to handle the problem of starvation of lower priority requests to the controller task. Also, it would be constructive to verify the effects of interleaving delays on the calls to the entry points. Doing so would confirm the fact that the algorithm does perform properly under more varying conditions.

There are many areas open for study in Ada. One of the more promising areas is that of generics. Future research in the area of generics and their use could come from studying object code size in a different vein than was pursued here. This research asserted conclusions about generics based on the generation of object code. Another direction might be to study generic structure and object code size together. It may be possible to derive a model which could predict the object code size of a program containing generics based on the structure of the generics alone. The model could then be modified to provide the prediction for other machines.

The Common Ada Missile Packages (CAMP), available at AFIT, provide a wide spectrum of source code containing generics.

There is also a good range of complexity within the CAMP software. This software and a good compiler (such as Verdix Ada 5.5) could be used to study and implement this topic. Results from this study would further the understanding of generics and their impact on embedded real time systems.

Conclusions

Working with Ada has been a bittersweet experience for me. Being weaned into programming on Ada, I have enormous respect for what it tries to be. Trying to master some of the complexities of the language has taught me many things. Although Ada is still relatively young, I feel it needs to mature more quickly in order to fulfill its purpose in the embedded systems world. There also needs to be a more concerted effort to get Ada into the mainstream of the DoD embedded systems. Without a strong commitment from the top, Ada will never receive the attention it deserves.

Appendix A: Test Data and Example Calculations

This appendix contains sample statistical calculations on some of the data gathered during the first part of this research. Additionally, all data on each compiler/machine combination tested is included. Each data set has a heading to identify the compiler and underlying machine used during the test. Three tests were performed on each compiler/machine combination.

The following example calculations were performed on the data from the Verdix Ada compiler on the Elxsi 6400. The calculations were performed on every data set in exactly the same manner. Table 1 in Chapter 3 summarized the results of statistical calculations on all the data gathered for these tests.

Calculations consisted of determining the expected values of a data set and then performing the Chi-Square Goodness-of-Fit Test. The hypothesis tested for each data set was:

H_0: Sample Data from uniform population distribution.

H_A: Sample Data not from uniform population distribution.

Using the first three data sets, which were generated on the Verdix Ada/Elxsi 6400 compiler/machine combination, observed values of each cell were calculated from a test sample of 30 program runs. Here the observed value for a particular cell is just the arithmetic sum of all sample observations. The expected

value for each data set is just the total number of observations in all cells multiplied by the probability of occurrence for the individual cells. Cell probabilities for Test 1 and Test 2 were .2, while the cell probabilities for Test 3 were .1. Then, using the equation $\Sigma(\text{Observed} - \text{Expected})^2/\text{Expected}$, the value of Chi-Square was calculated.

The number of degrees of freedom used in calculating Chi-Square for TEST 1, TEST 2, and TEST 3 were four, four, and nine. With the significance level, α, set at 0.05, the critical values of Chi-Square for the three tests were 9.488, 9.488, and 16.919 respectively (Barnes, 1988:351). Thus, if the data of a particular test yield a value of Chi-Square greater than the critical value, then there was ample reason to reject the null hypothesis. To not reject the null hypothesis meant that there was insufficient evidence to discredit it.

References used to understand and present the statistical information above included Kachigan (Kachigan, 1986:342-345,570) and Barnes (Barnes, 1988:322-325,351).

The sample calculations for the Verdix Ada/Elxsi 6400 combination indicated that the null hypothesis couldn't be rejected on any of the three tests. This then suggested that the scheduling algorithms in the Verdix Ada compiler were fair for the test configuration developed. All the other compiler/machine combinations tested caused the null hypothesis to be rejected on at least one test. Table 1 in Chapter 3 summarized the results of all calculations done for this appendix.

Verdix Ada/Elxsi 6400 Data (ICC)

TEST 1 Sample Data

TASK1	TASK2	TASK3	TASK4	TASK5
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
48	48	48	49	49
49	49	49	49	49
49	49	49	49	49
48	48	48	48	48
49	49	49	49	49
49	49	49	49	49
48	48	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
48	48	48	48	49
48	48	48	49	49
49	49	49	49	49
49	48	48	48	48
49	49	49	49	49
49	49	49	49	49
48	48	48	48	48
49	49	49	49	49
49	49	49	49	49

TEST 1 Expected Values

1464.8	1464.8	1464.8	1464.8	1464.8

TEST 1 Observed Values

1464	1463	1464	1466	1467

Chi-Squared = 0.007373

Since the critical value is 9.488, the null is not rejected.

TEST 2 Sample Data

TASK7	TASK8	TASK9	TASK10	TASK11
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
48	48	48	48	48
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
48	48	48	48	48
49	49	49	49	49
47	47	47	47	48
44	45	46	45	45
49	49	49	49	49
48	48	48	48	49
47	47	47	48	48
49	49	49	49	49
49	49	49	49	49
48	48	48	48	49
44	44	44	44	44
49	49	49	49	49
47	48	48	48	48
49	49	49	49	49
48	48	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
49	49	49	49	49
48	48	48	48	48

TEST 2 Expected Values

1451.4	1451.4	1451.4	1451.4	1451.4

TEST 2 Observed Values

1448	1450	1452	1452	1455

Chi-Square = 0.018741

Since the critical value is 9.488, the null is not rejected.

TEST 3 Sample Data

TSK11	TSK12	TSK13	TSK14	TSK15	TSK17	TSK18	TSK19	TSK20	TSK21
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	48	48	48	48	48
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	46	47	46	47	47
50	50	50	50	50	50	50	50	50	50
49	49	49	49	49	48	48	48	48	48
49	49	49	49	49	48	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	48	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
48	49	49	49	49	48	48	48	48	48
49	49	49	49	49	49	49	49	49	49
48	48	48	48	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
49	49	49	49	49	49	49	49	49	49
48	48	48	48	48	48	48	48	48	48
49	49	49	49	49	49	49	49	49	49

TEST 3 Expected Values

1466.6	1466.6	1466.6	1466.6	1466.6	1466.6	1466.6	1466.6	1466.6	1466.6

TEST 3 Observed Values

1468	1469	1469	1469	1470	1462	1465	1464	1465	1465

Chi-Square = 0.045275

Since the critical value is 16.919, the null is not rejected.

Verdix Ada/VAX 11/785 Data (SSC)

TEST 1 Sample Data

TASK1	TASK2	TASK3	TASK4	TASK5
397	398	399	399	399
347	347	348	344	348
331	328	332	331	329
296	297	298	299	299
363	361	363	363	363
323	323	324	321	321
370	372	372	370	371
315	315	314	317	317
336	335	335	334	335
335	335	337	337	337
371	374	374	372	374
398	398	400	399	400
233	235	234	235	235
165	166	164	166	165
106	107	108	108	108
152	150	150	152	151
165	165	163	166	166
237	237	238	239	238
337	333	336	338	335
355	353	356	356	357
396	393	397	397	396
397	396	397	398	396
380	381	379	381	380
396	397	396	398	399
316	314	318	316	318
294	294	293	293	295
330	331	331	331	331
178	178	178	178	178
261	262	262	260	263
341	340	341	342	340

TEST 1 Expected Values

9231.4	9231.4	9231.4	9231.4	9231.4

TEST 1 Observed Values

9221	9215	9237	9240	9244

Chi-Square = 0.069459

Since the critical value is 9.488, the null is not rejected.

TEST 2 Sample Data

TASK7	TASK8	TASK9	TASK10	TASK11
323	322	322	322	322
288	289	288	289	289
272	272	272	272	273
271	271	271	271	271
258	258	258	259	258
282	282	282	282	283
267	267	267	268	268
272	271	272	273	273
228	228	228	227	229
243	244	243	244	244
214	214	214	214	215
255	255	256	253	256
279	279	279	278	278
258	258	258	258	257
268	267	268	268	268
248	248	249	249	249
237	237	237	238	238
297	296	297	297	297
271	271	271	271	272
292	292	292	292	293
283	282	283	284	283
241	242	241	242	242
236	236	237	235	237
280	281	281	281	281
303	304	303	303	304
294	294	294	293	294
296	296	296	297	297
261	261	261	261	262
253	253	253	253	252
294	295	295	294	295

TEST 2 Expected Values

8069	8069	8069	8069	8069

TEST 2 Observed Values

8064	8065	8068	8068	8080

Chi-Square = 0.020325

Since the critical value is 9.488, the null is not rejected.

TEST 3 Sample Data

TSK11	TSK12	TSK13	TSK14	TSK15	TSK17	TSK18	TSK19	TSK20	TSK21
222	222	223	222	223	213	213	213	213	213
176	175	176	176	177	160	159	159	159	159
187	187	188	188	188	176	176	174	175	176
198	197	199	199	198	179	179	179	179	179
204	203	204	204	202	190	190	190	190	191
187	187	187	187	188	171	170	171	170	170
197	198	198	199	199	184	183	183	184	183
175	174	176	175	176	163	164	164	164	164
201	201	200	201	201	184	182	183	183	181
198	198	199	198	198	186	186	185	186	186
193	194	194	194	195	179	179	178	178	180
187	190	189	190	191	166	166	167	166	166
194	195	195	195	196	177	175	176	176	176
193	192	193	193	194	177	177	177	178	176
176	177	176	177	177	156	158	157	157	156
211	211	212	211	212	194	193	193	193	194
207	207	208	207	208	193	192	193	193	194
207	207	207	208	208	189	189	188	189	189
166	167	166	167	165	155	154	155	154	155
201	202	202	202	201	183	185	184	184	185
83	83	83	83	84	78	78	78	78	78
74	75	75	74	75	67	67	67	66	67
102	102	102	102	102	90	91	90	91	91
181	183	184	184	184	164	164	164	164	163
214	215	214	215	215	193	193	194	193	193
206	204	208	206	208	189	188	190	189	190
220	221	219	221	220	202	203	204	204	205
223	224	224	224	224	209	210	209	210	210
230	230	230	230	230	210	210	210	210	209
180	179	180	181	181	167	167	167	167	167

TEST 3 Expected Values

5375.3 5375.3 5375.3 5375.3 5375.3
5375.3 5375.3 5375.3 5375.3 5375.3

TEST 3 Observed Values

5593	5600	5611	5613	5620	5144	5141	5142	5143	5146

Chi-Square = 100.3077

Since the critical value is 16.919, the null is rejected.

DEC Ada/VAX 8650 Data (ASD/SCEL)

TEST 1 Sample Data

TASK1	TASK2	TASK3	TASK4	TASK5
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
245	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249

TEST 1 Expected Values

7470	7470	7470	7470	7470

TEST 1 Observed Values

7470	7470	7470	7470	7470

Chi-Square = 0.0

Since the critical value is 9.488, the null is not rejected.

TEST 2 Sample Data

TASK7	TASK8	TASK9	TASK10	TASK11
249	249	249	249	250
249	249	249	249	249
249	249	249	249	249
249	249	249	249	250
249	249	249	249	249
249	249	249	249	250
249	249	249	249	249
249	249	249	249	249
249	249	249	249	250
249	249	249	249	249
249	249	249	250	250
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	250	250	250	250
249	249	249	249	249
249	249	249	249	250
249	249	249	250	250
249	249	249	249	249
249	250	250	250	250
249	249	249	249	249
249	249	250	249	250
249	249	249	249	249
249	249	249	250	250
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	250

TEST 2 Expected Values

7474.2	7474.2	7474.2	7474.2	7474.2

TEST 2 Observed Values

7470	7472	7473	7475	7482

Chi-Square = 0.011399

Since the critical value is 9.488, the null is not rejected.

TEST 3 Sample Data

TSK11	TSK12	TSK13	TSK14	TSK15	TSK17	TSK18	TSK19	TSK20	TSK21
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	233	173	172	173
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167
249	249	250	250	250	250	250	167	167	167

TEST 3 Expected Values

6747	6747	6747	6747	6747	6747	6747	6747	6747	6747

TEST 3 Observed Values

7470	7470	7500	7500	7500	7500	7483	5016	5015	5016

Chi-Square = 1904.215

Since the critical value is 16.919, the null is rejected.

DEC Ada/VAX 11/780 Data (ISL)

TEST 1 Sample Data

TASK1	TASK2	TASK3	TASK4	TASK5
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	250	250	250	250
249	249	250	250	250
249	250	250	250	250
249	249	250	250	250
250	250	250	250	250
249	249	250	250	250
249	249	250	250	250
249	249	250	250	250
249	249	250	250	250
249	249	250	250	250
245	249	250	250	250
249	249	250	250	250
249	249	249	249	250
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249
250	250	250	250	250
249	249	249	249	250
249	249	249	249	249
249	249	249	249	249
249	249	249	249	249

TEST 1 Expected Values

7478.6	7478.6	7478.6	7478.6	7478.6

TEST 1 Observed Values

7468	7474	7483	7483	7485

Chi-Square = 0.028508

Since the critical value is 9.488, the null is not rejected.

TEST 2 Sample Data

TASK7	TASK8	TASK9	TASK10	TASK11
248	248	246	246	246
249	249	248	248	248
249	248	247	246	246
249	249	247	247	248
249	248	247	247	247
249	248	247	247	247
247	247	247	247	247
249	249	247	247	248
249	247	247	246	247
249	247	247	247	247
248	248	246	246	247
249	248	247	247	247
249	248	246	246	247
249	249	247	247	248
249	248	256	246	247
249	249	247	247	248
250	248	248	248	247
249	248	247	247	247
248	248	248	246	247
249	249	247	247	248
249	247	247	246	247
249	249	248	247	248
249	247	248	248	247
249	248	247	247	247
249	247	247	247	247
249	248	247	248	248
248	247	246	246	247
249	248	247	248	248
248	248	247	247	247
248	248	247	247	247

TEST 2 Expected Values

7429.2	7429.2	7429.2	7429.2	7429.2

TEST 2 Observed Values

7463	7440	7420	7406	7417

Chi-Square = 0.273354

Since the critical value is 9.488, the null is not rejected.

TEST 3 Sample Data

TSK11	TSK12	TSK13	TSK14	TSK15	TSK17	TSK18	TSK19	TSK20	TSK21
249	249	250	250	250	201	196	197	197	197
249	249	249	249	250	205	197	196	196	197
249	249	249	249	250	240	187	186	186	186
249	249	250	250	250	203	196	197	197	197
247	247	247	247	247	229	183	183	183	183
249	249	250	250	250	200	197	197	197	198
249	249	250	250	250	235	189	189	189	189
249	249	250	250	250	239	188	187	187	188
249	249	250	250	250	242	187	186	186	187
245	245	245	245	245	214	185	185	185	185
249	249	249	249	249	226	191	190	191	191
249	249	250	250	250	224	191	192	191	192
249	249	249	250	250	238	187	187	188	188
249	249	250	250	250	242	186	187	187	187
243	243	244	244	244	209	190	189	189	189
243	244	244	244	244	224	181	181	181	181
249	249	249	249	249	226	191	190	191	191
249	249	249	250	250	240	187	187	187	188
245	245	245	246	246	230	183	183	182	183
249	249	249	249	250	238	188	188	187	188
248	248	249	249	249	243	186	186	186	186
243	244	244	244	244	219	181	180	180	181
249	249	249	250	250	237	188	188	188	188
249	249	250	250	250	239	188	188	187	188
247	247	248	248	248	224	188	187	187	188
249	249	249	249	249	245	186	187	186	187
249	249	250	250	250	221	193	192	192	192
249	249	249	250	250	242	187	186	186	187
249	249	249	249	249	238	187	187	187	188
247	247	247	247	247	234	186	186	185	186

TEST 3 Expected Values

6668.3	6668.3	6668.3	6668.3	6668.3	6668.3	6668.3	6668.3	6668.3	6668.3

TEST 3 Observed Values

7437	7439	7452	7457	7460	6847	5650	5644	5641	5656

Chi-Square = 1086.642

Since the critical value is 16.919, the null is rejected.

JANUS Ada/Z-248 Data

TEST 1 Sample Data

TASK1	TASK2	TASK3	TASK4	TASK5
24	24	24	24	24
25	24	24	24	25
25	25	25	24	25
24	24	24	23	24
24	24	24	24	24
24	24	24	24	25
25	24	24	24	25
24	24	24	24	24
24	24	24	24	25
24	24	24	24	24
25	24	24	24	25
24	24	24	24	24
24	24	24	24	24
24	24	24	24	25
24	24	24	24	25
24	24	24	23	24
25	24	24	24	25
24	24	24	24	25
25	24	24	24	25
24	24	24	23	24
24	24	24	24	25
24	24	24	24	25
24	24	24	24	24
24	24	24	24	24
24	24	24	24	25
25	24	24	24	25
24	24	24	24	24
24	24	24	23	24
25	25	25	24	25
24	24	24	23	24

TEST 1 Expected Values

724.6	724.6	724.6	724.6	724.6

TEST 1 Observed Values

728	722	722	717	734

Chi-Square = 0.236268

Since the critical value is 9.488, the null is not rejected.

TEST 2 Sample Data

TASK7	TASK8	TASK9	TASK10	TASK11
17	16	1	1	2
16	16	1	1	2
16	16	1	1	2
17	16	1	1	2
17	16	1	1	2
17	16	1	1	2
17	16	1	1	2
16	16	1	1	2
17	16	1	1	2
16	16	1	1	2
16	16	1	1	2
17	16	1	1	2
16	16	1	1	2
17	16	1	1	2
16	16	1	1	2
16	16	1	1	2
16	16	1	1	2
17	16	1	1	2
17	16	1	1	2
17	16	1	1	2
16	16	1	1	2
17	17	1	1	2
16	16	1	1	2
16	16	1	1	2
16	16	1	1	2
16	16	1	1	2
17	16	1	1	2
16	16	1	1	2
16	16	1	1	2
16	16	1	1	2

TEST 2 Expected Values

218.8	218.8	218.8	218.8	218.8

TEST 2 Observed Values

493	481	30	30	60

Chi-Square = 1098.916

Since the critical value is 9.488, the null is rejected.

TEST 3 Sample Data

TSK11	TSK12	TSK13	TSK14	TSK15	TSK17	TSK18	TSK19	TSK20	TSK21
17	17	17	17	16	5	4	1	1	2
17	17	17	17	16	5	4	1	1	2
17	17	17	17	16	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	16	5	4	1	1	2
18	18	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
18	17	17	17	17	5	4	1	1	2
17	17	17	17	16	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	16	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
18	17	17	17	17	5	4	1	1	2
17	17	17	17	16	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	16	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
18	18	17	17	17	5	4	1	1	2
17	17	17	17	16	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2
17	17	17	17	17	5	4	1	1	2

TEST 3 Expected Values

293.7	293.7	293.7	293.7	293.7	293.7	293.7	293.7	293.7	293.7

TEST 3 Observed Values

514	512	510	510	501	150	120	30	30	60

Chi-Square = 1624.937

Since the critical value is 16.919, the null is rejected.

Appendix B: Program Output of Part 2

This appendix contains the output from the program written for part two of this research. The output of the program records execution of the implied priority scheme for 10 combinations of available entry points and calls made. The following list contains the compiler/machine combinations that this test was successfully run on:

COMPILER	OPERATING SYS	MACHINE
Verdix 5.5	UNIX BSD 4.3	ELXSI 6400
Verdix 5.41	UNIX BSD 4.3	VAX 11/780
DEC Ada 4.6	VMS 1.4	VAX 11/780
DEC Ada 4.7	VMS 1.5	VAX 8650
JANUS 2.02	MS-DOS 3.2	Z-248

Asterisks mark the points in the tests where entry point calls were accepted and executed.

```
Output TEST_1
=========================================
Ready to accept call at HIGHEST Priority!
Executing HIGHEST Priority Request!          *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Executing MIDDLE Priority Request!           *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
Executing LOWEST Priority Request!           *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!
```

```
Output TEST_2
=========================================
Ready to accept call at HIGHEST Priority!
Executing HIGHEST Priority Request!          *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Executing MIDDLE Priority Request!           *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!


Output TEST_3
=========================================
Ready to accept call at HIGHEST Priority!
Executing HIGHEST Priority Request!          *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
Executing LOWEST Priority Request!           *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!


Output TEST_4
=========================================
Ready to accept call at HIGHEST Priority!
Executing HIGHEST Priority Request!          *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!

Output TEST_5
=========================================
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Executing MIDDLE Priority Request!           *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
Executing LOWEST Priority Request!           *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!
```

```
Output TEST_6
=============================================
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Executing MIDDLE Priority Request!              *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!

Output TEST_7
=============================================
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
Executing LOWEST Priority Request!              *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!


Output TEST_8
=============================================
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!


Output TEST_9
=============================================
Ready to accept call at HIGHEST Priority!
Executing HIGHEST Priority Request!             *
Ready to accept call at HIGHEST Priority!
Executing HIGHEST Priority Request!             *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
Executing LOWEST Priority Request!              *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!
```

```
Output TEST_10
=======================================
Ready to accept call at HIGHEST Priority!
Executing HIGHEST Priority Request!          *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
Executing LOWEST Priority Request!           *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
Executing LOWEST Priority Request!           *
Ready to accept call at HIGHEST Priority!
Ready to accept call at MIDDLE Priority!
Ready to accept call at LOWEST Priority!
No calls pending! Execution TERMINATED!!
```

Appendix C: Graphical Results of Part 3

This appendix contains graphical data on the results of comparing object code generated by generic programs and their explicitly coded counterparts. The data used to generate the graphs is also included. Each graph represents the results from a single compiler/machine combination and six test programs.

Verdix Ada/ELEXSI 6400 Data (Bytes)

	GENERIC	EXPLICIT
1.	67884	67042
2.	69981	69171
3.	63355	63374
4.	71664	70795
5.	80264	79416
6.	65405	65414

Verdix Ada/VAX 11/785 Data (Bytes)

	GENERIC	EXPLICIT
1.	22528	19456
2.	24576	21504
3.	18432	18432
4.	24576	24576
5.	28672	27648
6.	19456	19456

DEC Ada/VAX 11/780 Data (Bytes)

	GENERIC	EXPLICIT
1.	2861	2818
2.	2712	3707
3.	2500	2467
4.	4813	4730
5.	6652	6684
6.	2450	2418

DEC Ada/VAX 8650 Data (Bytes)

	GENERIC	EXPLICIT
1.	2349	2305
2.	3224	3809
3.	2500	2467
4.	4813	4730
5.	6652	6684
6.	2450	2420

JANUS Ada/Z-248 Data (Bytes)

	GENERIC	EXPLICIT
1.	20600	10669
2.	23167	16677
3.	22439	13848
4.	23874	19069
5.	25301	26317
6.	22618	14720

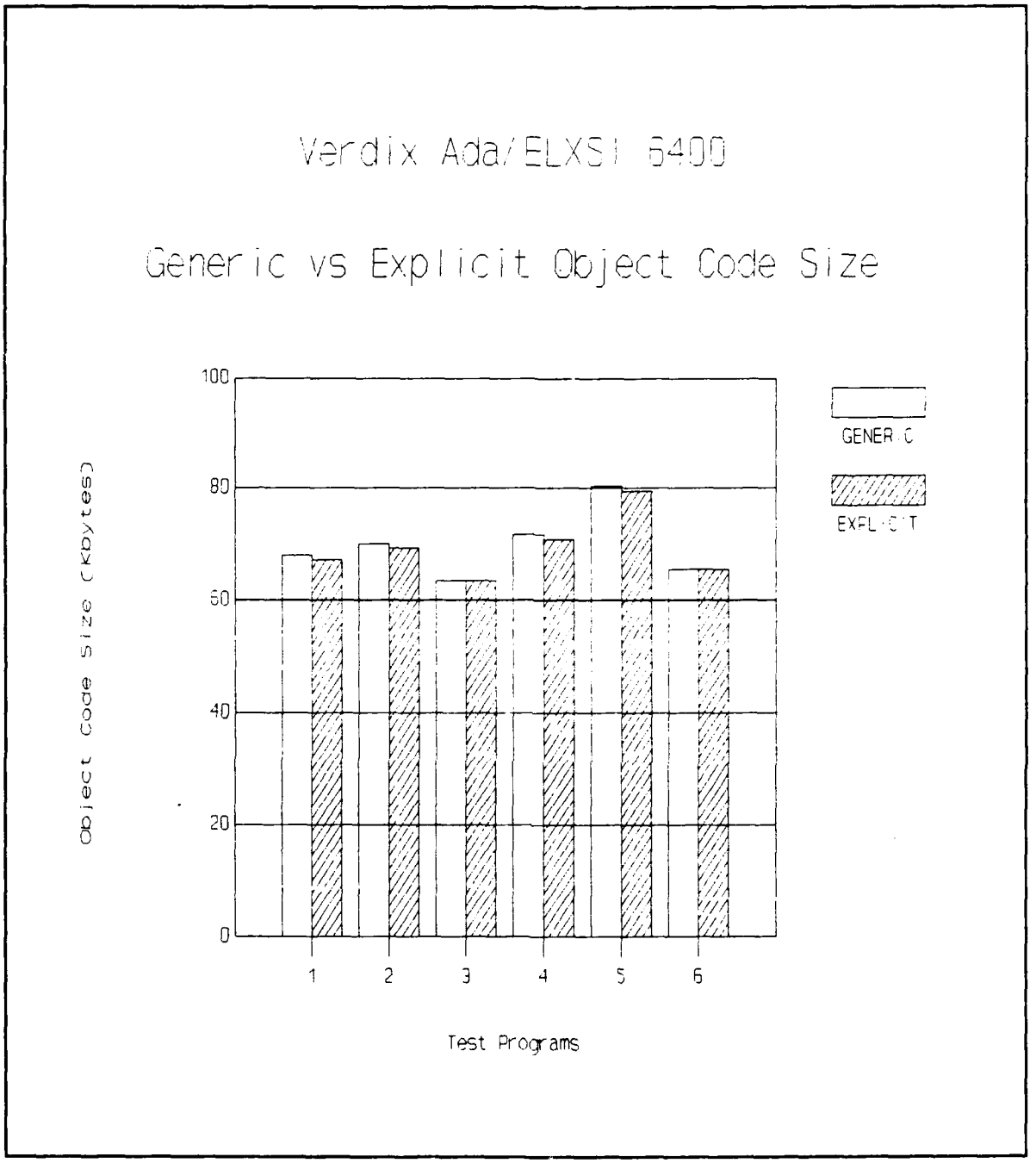

Figure 1 Verdix Ada/ELXSI 6400 (ICC) Results

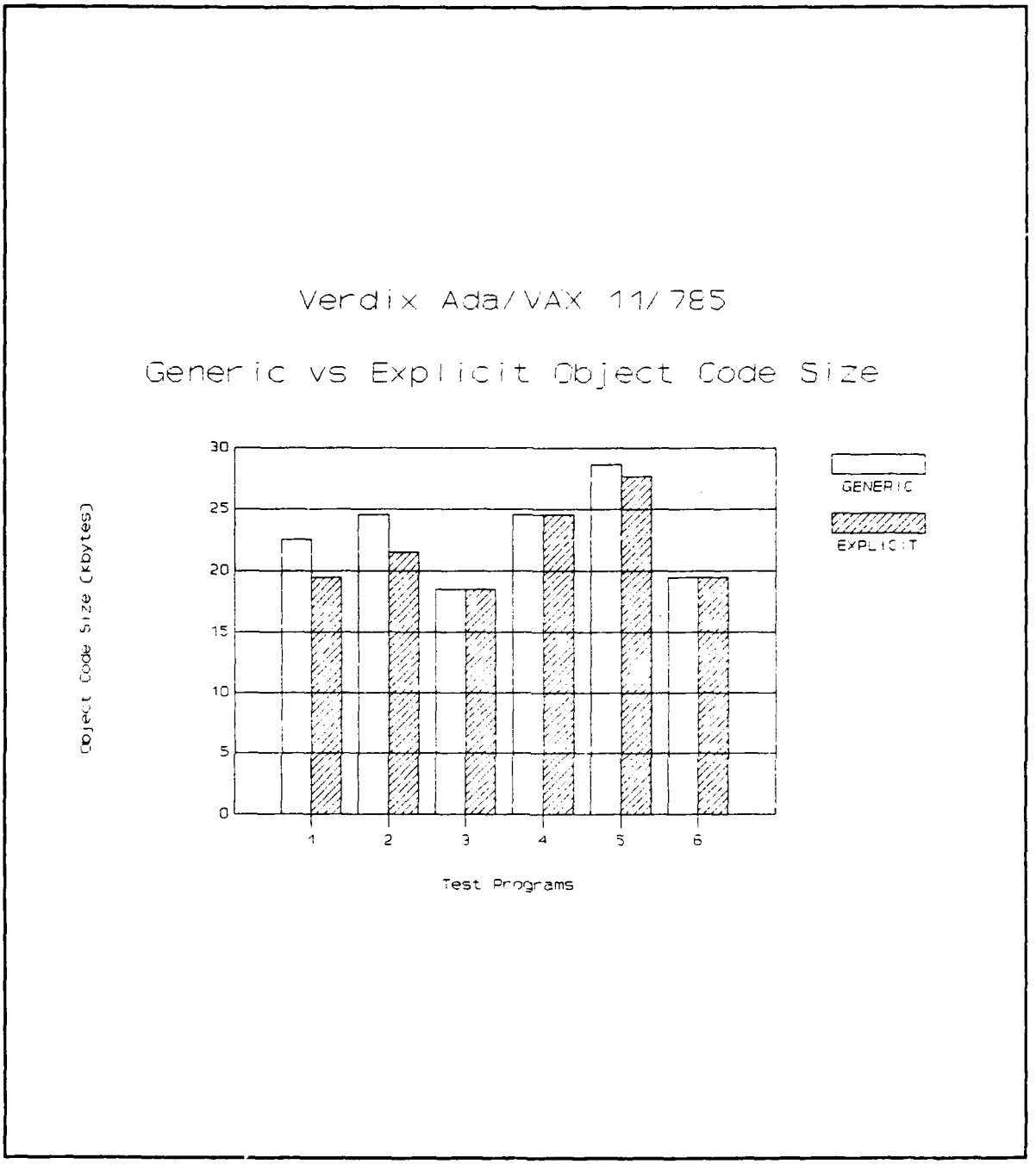

Figure 2 Verdix Ada/VAX 11/785 (SSC) Results

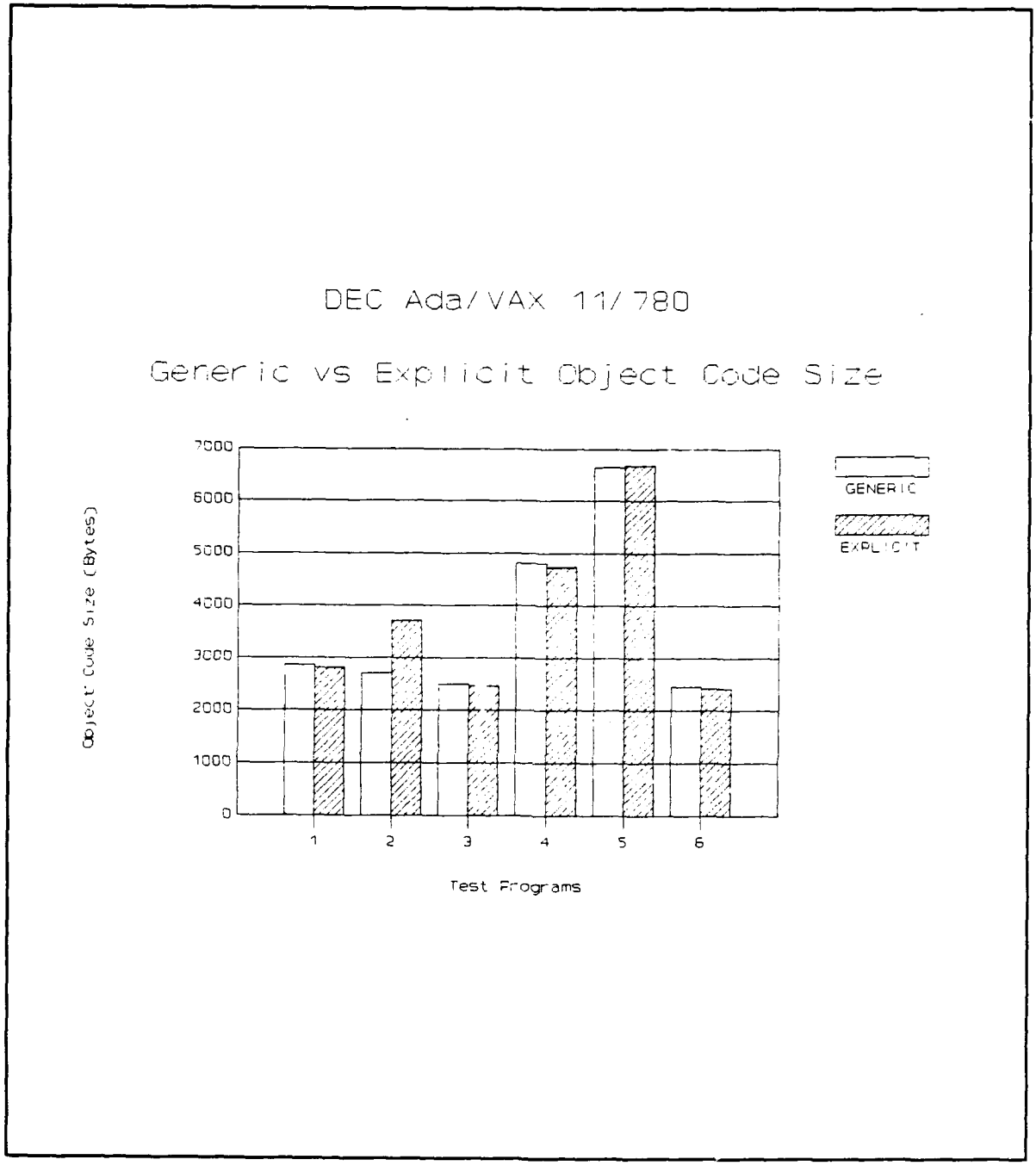

Figure 3 DEC Ada/VAX 11/780 (ISL) Results

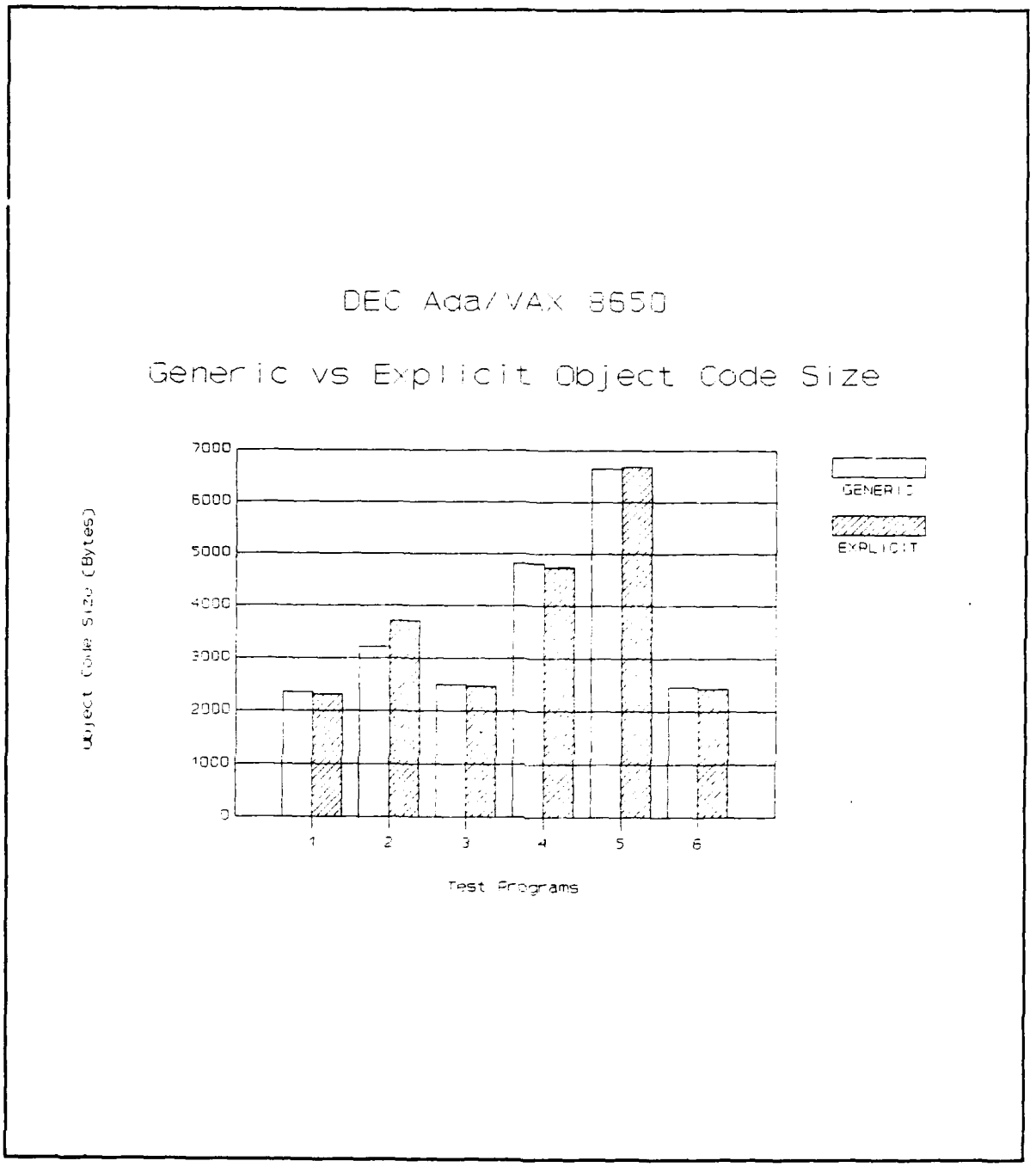

Figure 4 DEC Ada/VAX 8650 (ASD/SCEL) Results

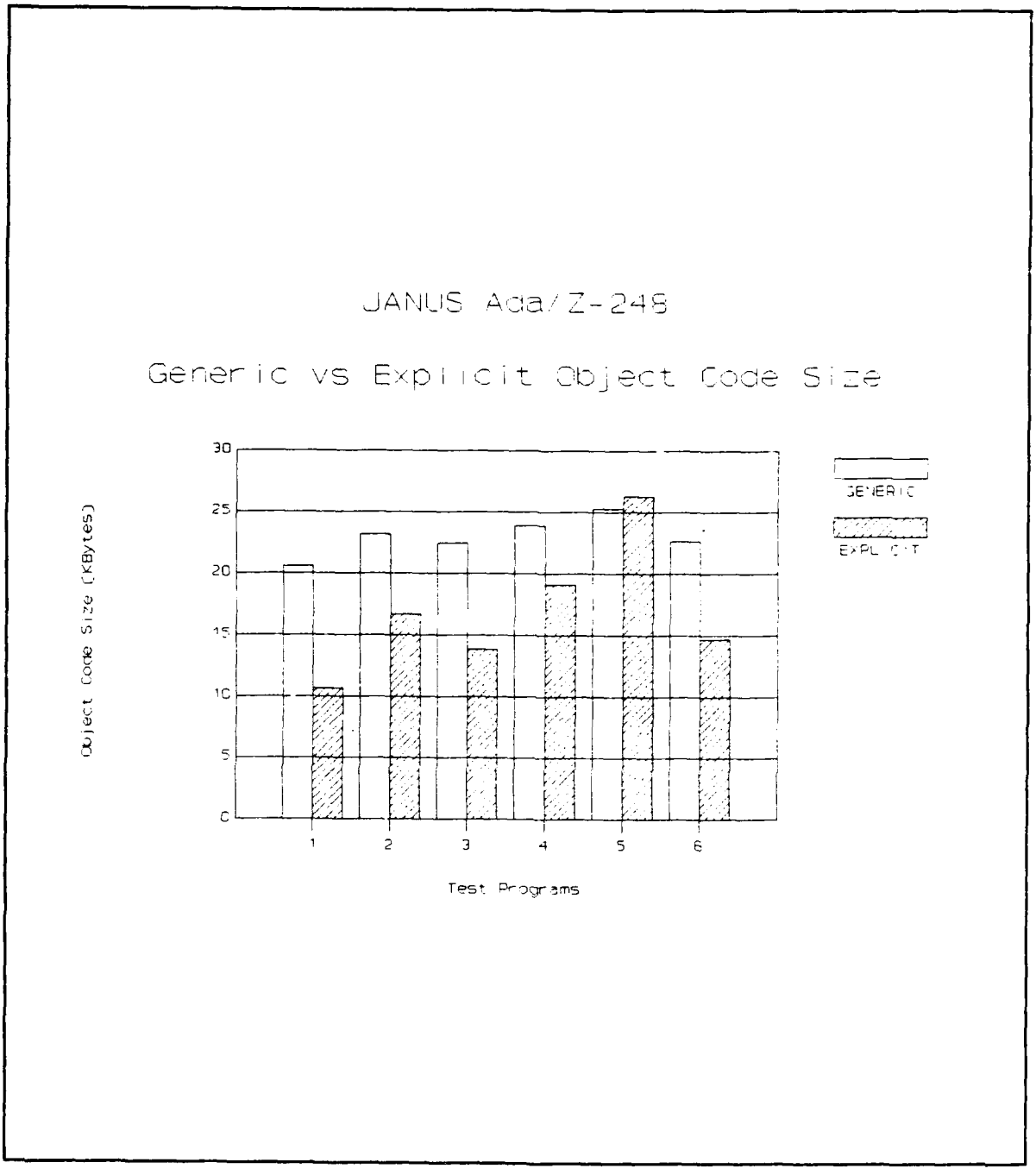

Figure 5 JANUS Ada/Z-248 Results

Bibliography

Ada Board. "Ada Board Report: Executive Summary, February 1988", Ada Letters, Volume VIII, Number 4:47-68, July/August 1988.

Aho, Alfred V. Compilers: Principles, Techniques, and Tools. Addison-Wesley Publishing Company, Reading, Mass. 1986.

AREWG. "A Framework for Describing Ada Run-Time Environments", Ada Letters, Volume VIII, Number 3:51-68, May/June 1988.

ARINC. "Guidance for Using the Ada Programming Language in Avionics Systems", Draft 2 of Project Paper 613, Aeronautical Radio Inc., Annapolis, Maryland. May 1987.

Booch, Grady. Software Engineering with Ada (Second Edition). Menlo Park CA: The Benjamin/Cummings Publishing Company Inc., 1987.

Burns, A. "Real Time Ada Issues", Ada Letters, Volume VII, Number 6:43-46, (Fall 87).

Department of Defense. Requirements for High Order Computer Programming Language (IRONMAN). 1977.

Department of Defense. Military Standard: Ada Programming Language Reference Manual. MIL-STD-1815A. Washington: Department of Defense, 22 January 1983.

Department of Defense. Use of Ada in Weapons Systems. DOD Directive 3405.2. Washington: Government Printing Office, 30 March 1987.

Dykstra E.W. "DOD-1: The Summing Up," Sigplan Notices, Vol 13, Number 7:21-26, July 1978.

Elbert, Theodore F. Embedded Programming in Ada. VanNostrand Reinhold Company Inc., 1986.

Evans, Bobby. Head, Ada Language Control Facility. Official Correspondence. Proposed Thesis Topics. July 1987.

Fisher, David A. "DOD's Common Programming Language Effort," Computer, Vol II, Number 3:24-33, March 78.

Gehani, Narain. Ada Concurrent Programming, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1984.

----- UNIX Ada Programming, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1987.

Goodenough, John B. "The Ada Compiler Validation Capability," Computer, Vol 14, Number 6:57-64, June 1981.

Goodenough, John B. The Ada Compiler Validation Capability Implementors' Guide: Version 1. SofTech, Inc., Waltham, MA, December 1986.

Herr, Christine S. et al. "Compiler Validation and Reusable Ada Parts for Real-Time Embedded Applications," Executive Summary of Common Ada Missle Packages, Ada Letters Volume VIII, Number 5:75-86, Sept/Oct 1988.

Hoare, C.A.R. "Communicating Sequential Processes'" Communications of the ACM, Vol 21, Number 8:666-676, Aug 1978.

Hoare, C.A.R. "The Emperor's Old Clothes," Communications of the ACM, Vol 24, Number 2:75-83, Feb 1981.

Howatt, Capt James. Assistant Professor, Department of Electrical Engineering, Air Force Institute of Technology (AU). Personal Interview. Wright-Patterson AFB OH. 4 May 1988.

Howden, William E. Functional Program Testing and Analysis, McGraw-Hill Book Company, New York, 1987.

Jensen, Randall W. and Charles C. Tonies. Software Engineering, Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1979.

King, Capt Dave. Computer Resources System Engineer, ASD/YFEA. Telephone Interview. Aeronautical Systems Division, WPAFB, OH. 26 Sep 1988.

Ledgard Henry F. and Andrew Singer. "Scaling Down Ada (Or Towards A Standard Ada Subset)", Communication of the ACM, Vol 25, Number 2:121-125, February 1982.

Locke C. Douglass and David R. Vogel. "Problems in Ada Run-time Scheduling," Ada Letters, Volume VII, Number 6:51-53 (Fall 1987).

Lyon, Major Bob. Chief, Core Avionics Division, ASD/YFAC. Telephone Interview. Aeronautical Systems Division, WPAFB, OH. 19 Sep 1988.

McCormick, Frank. "Scheduling Difficulties of Ada In the Hard Real-Time Environment", Ada Letters, Volume VII, Number 6: 49-50 (Fall 1987).

Peterson James L. and Abraham Silberschatz. Operating Systems Concepts (Second Edition), Addison-Wesley Publishing Company, 1985.

Pressman, Roger S. Software Engineering: A Practitioners Approach (Second Edition), McGraw-Hill, Inc., New York. 1987.

Port, Otis et al. "The Software Trap: Automate or Else", Business Week, Number 3051: 142-154 (9 May 1988).

Shaw, Mary et al. "TARTAN: Language Design for the IRONMAN Requirement: Reference Manual", SIGPlan Notices, Volume 13, Number 9:36-58, September 1978.

Shumate, Kenneth C. Understanding Ada, Harper and Row Publishers, New York, 1984.

----- Understanding Concurrency in Ada, McGraw-Hill Book Company, New York, 1988.

Softech. "Program Office Guide to Ada Edition 1", Softech Inc., Waltham, MA., September 1986. Electronic Systems Division TR-86-282.

Wichmann, Brian A. "Is Ada Too Big? A Designer Answers the Critics", Communications of the ACM, Volume 27, Number 2:98-103, February 1984.

Wilson, Steven. Tecnical Director, Ada Language Control Facility, ASD/SCEL. Telephone Interview. Aeronautical Systems Division, WPAFB, OH. 15 Feb 1989.

Woffinden, Duard S., Assistant Professor, Department of Electrical Engineering, Air Force Institute of Technology (AU). Lecture Notes. Wright-Patterson AFB OH. Fall 1987.

VITA

Captain Robert H. Tippett [redacted] he enlisted in the U.S. Navy. He was honorably discharged from active duty in 1978. He entered Miami University at Oxford, Ohio in 1979 to study Aeronautical Science. He enlisted in the Ohio Air National Guard where he served from 1979 to 1984 as a Radio Communications Technician. In 1983 he received the degrees of Bachelor of Science in Aeronautics and Bachelor of Arts in Aeronautics/Mathematics. He re-entered active duty in February 1984 and was subsequently commissioned as a second lieutenant through the U.S. Air Force Officer Training School at Lackland AFB, Texas. He then attended Pennsylvania State University on an Air Force Institute of Technology Scholarship and received the degree of Bachelor of Science in Meteorology in 1985. From May 1985 to May 1987 he served as a Wing Weather Officer at Detachment 26, 26th Weather Squadron, Grissom AFB, Indiana. There he supported the missions of the 305th Air Refueling Wing and 305th Combat Support Group. Currently a student, in residence, at the Air Force Institute of Technology studying for a Masters Degree in Computer Science from the Department of Electrical Engineering.

[redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/89M-3	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DoD Ada Validation Facility 8b. OFFICE SYMBOL (If applicable) ASD/SCEL	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB, OH 45433-6583	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Conformity Issues of Ada Tasking and the Effects of Ada Generics on Object Code Size (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Robert H. Tippett, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1989 March 17	98

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Compiler Testing, Computer Program Verification, Computer Programming, Compilers
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
Thesis Chairman: James W. Howatt, Major, USAF
Assistant Professor of Electrical Engineering and Computer Science

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Major James W. Howatt	22b. TELEPHONE (Include Area Code) 513-255-6913 / 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

19. -cont- This thesis deals with compiler testing and program verification. Specifically, three areas are addressed concerning Ada compilers. First, several compilers are assessed to determine the fairness of their task scheduling algorithms. Next, conditional entry calls are used to develop an implied priority scheme within a program. Finally, the effects of using generics is explored. The effects of generics on object code size is determined.

The first tests dealing with Ada tasking examine the effects of the rendezvous on the fairness of the scheduling algorithms. Three different tests are used to assess the effects of the rendezvous. Autonomous tasks are run by themselves, as are tasks with rendezvous. Then these two cases are combined to test there effects on the compiler operation.

The second test is desinged to show that an implied priority scheme can be implemented using the conditional entry call. The calls are nested three levels deep to confirm the operation of the designed algorithm.

The final test explores the effects of using generics on object code size. Test programs are coded with generics and explicitly. Then the compiled programs are compared with respect to object code. The results are presented in tables and graphically.

The results of these three tests showed that most compilation systems do not perform well when the hybrid test case for testing scheduling fairness is used. Also, it was shown that the implied priority scheme could be implemented with nested conditional entry calls. Finally the research showed that it is not necessarily the case that Ada generics cause the inflation of object code size.